NASA Technical Memorandum 88288

Manual for *GetData* Version 3.1

A FORTRAN Utility Program for Time History Dat

Richard E. Maine

October 1987

NASA
National Aeronautics and
Space Administration

NASA Technical Memorandum 88288

Manual for *GetData* Version 3.1

A FORTRAN Utility Program for Time History Datɛ

Richard E. Maine
Ames Research Center, Dryden Flight Research Facility, Edwards, California

1987

National Aeronautics and
Space Administration
Ames Research Center

Contents

SUMMARY 1

1 Introduction 1

2 User's Guide to *GetData* 2

2.1 Running *GetData* 2
2.2 Entering *GetData* Commands 3
2.3 *Help* Command 4
2.4 Basic Operation 5
2.5 Controlling the Output Frame Times 7
2.6 Merging and Splicing Input Files 9
2.7 Applying Time Skews 11
2.8 Interpolating in Time 13
2.9 Selecting and Defining Signals 14
2.10 Showing Signal Definitions 17
2.11 Automating Command Sequences 17
2.12 Running System Commands From *GetData* 19
2.13 Specifying File Formats 19

3 Programmer's Guide to *GetData* 21

3.1 Calculated Function Modules 21
3.2 Filter Module 22
3.3 File Interface Modules 23
3.4 File Formats 25
3.5 System Dependencies 27
3.6 Specific Conversions 31
3.6.1 VAX-VMS Conversion 31
3.6.2 UNIX Conversion 32

Appendix A—Help Files 34

A.1 Program Help File 34
A.2 Command Help Files 36
A.2.1 *Copy* Command 36
A.2.2 *Do* Command 38
A.2.3 *Help* Command 40
A.2.4 *Method* Command 41
A.2.5 *Quit* Command 43
A.2.6 *Read* Command 44
A.2.7 *Show* Command 46
A.2.8 *Signals* Command 47
A.2.9 *Skew* Command 51
A.2.10 *Sys* Command 53
A.2.11 *Write* Command 54
A.3 Topic Help Files 56
A.3.1 Calculations 56
A.3.2 CPUTime 59
A.3.3 FileInterFace 63

A.3.4 Version 66
A.4 Calculated Function Subroutine Help Files 70
A.4.1 Subroutine *AllocateCFx* 70
A.4.2 Subroutine *ActivateCFx* 73
A.4.3 Subroutine *DoCFx* 76
A.5 Filter Subroutine Help Files 79
A.5.1 Subroutine *AllocateFilt* 79
A.5.2 Subroutine *ActivateFilt* 81
A.5.3 Subroutine *ReMapFilt* 83
A.5.4 Subroutine *DoFilt* 85
A.6 File Read Subroutine Help Files 87
A.6.1 Function *OpenR* 87
A.6.2 Subroutine *RSigs* 89
A.6.3 Subroutine *SigsR* 91
A.6.4 Subroutine *ChansR* 93
A.6.5 Subroutine *RewR* 95
A.6.6 Function *FSeek* 97
A.6.7 Function *FRead* 99
A.6.8 Subroutine *CloseR* 101
A.7 File Write Subroutine Help Files 103
A.7.1 Function *OpenW* 103
A.7.2 Subroutine *FWrite* 106
A.7.3 Subroutine *CloseW* 108
A.8 File Format Help Files 110
A.8.1 ASCII 1 Format 110
A.8.2 Compressed 2 Format 115
A.8.3 List 1 Format 118
A.8.4 Uncompressed 1 Format 119
A.8.5 Uncompressed 2 Format 121

Appendix B—Sample Calculation Routines **123**
B.1 Sample Calculated Function Module 123
B.1.1 Subroutine *AllocateCF1* 123
B.1.2 Subroutine *ActivateCF1* 125
B.1.3 Subroutine *DoCF1* 126
B.2 Sample Filter Module 127
B.2.1 Subroutine *AllocateFilt* 127
B.2.2 Subroutine *ActivateFilt* 130
B.2.3 Subroutine *ReMapFilt* 131
B.2.4 Subroutine *DoFilt* 132
B.2.5 Subroutine *Low3F* 134
B.2.6 Subroutine *NotchF* 137

References **139**

Index to User's and Programmer's Guides **140**

SUMMARY

This report documents version 3.1 of the *GetData* computer program. *GetData* is a utility program for manipulating files of time history data, that is, data giving the values of parameters as functions of time. The most fundamental capability of *GetData* is extracting selected signals and time segments from an input file and writing the selected data to an output file. Other capabilities include converting file formats, merging data from several input files, time skewing, interpolating to common output times, and generating calculated output signals as functions of the input signals.

This report also documents the interface standards for the subroutines used by *GetData* to read and write the time history files. All interface to the data files is through these subroutines, keeping the main body of *GetData* independent of the precise details of the file formats. Different file formats can be supported by changes restricted to these subroutines. Other computer programs conforming to the interface standards can call the same subroutines to read and write files in compatible formats.

1 Introduction

Aircraft flight test and research projects often generate large amounts of computer data. A single flight of a complex vehicle typically generates several hundred megabytes of data. A single flight project may involve several hundred flights, and a dozen active flight projects may be in progress at a major flight test or research site. This gives a total volume on the order of a terabyte of data to be managed.

The overwhelming majority of these data can be classified as time history data, that is, data showing the values of various parameters (signals) as functions of time. The parameter values are usually sampled and recorded at regular time intervals. Different parameters on the same vehicle can have different sample rates, typically ranging between 1 and 1000 samples/sec. In some cases, parameters are sampled at irregular time intervals; such asynchronous sampling is relatively rare (but not unheard of) in current data systems.

The *GetData* program is a utility for performing several functions fundamental to files of time history data. The most fundamental capability of *GetData* is extracting selected signals and time segments from an input file and writing the selected data to an output file. Other capabilities include converting file formats, merging data from several input files, time skewing, interpolating to common output times, and generating calculated output signals as functions of the input signals.

Time history data are used, manipulated, and exchanged among dozens of computer programs. Until recently each program was typically written to use a specific file format for time history data. There was only minimal coordination of these file formats.

This proliferation of incompatible file formats necessitated numerous program patches to read and write different formats; conversion programs were also written to translate file formats. Although each patch or conversion program required relatively little effort, the large number of them and the large volume of data involved meant that the total effort expended was substantial.

An obvious approach to dealing with this problem is to minimize the number of incompatible formats used, adopting a small number of formats as supported standards at a site. This approach can substantially reduce, but cannot eliminate, the problem. The volume of existing files is too large for it to be practical to reformat them all. We do not always have the option to specify the file formats used by commercial programs or received from other sites.

The *GetData* program addresses this problem by modularization. All code that is dependent on for the interface between these file access subroutines and the rest of the program. This allows the same subroutines to be used in any program conforming to the interface standard. Support of a new file format then requires only that a single version of the access routines be written for that format. This requires relinking each pertinent program with the new access routines but requires no source code changes outside the access routines.

We have recommended that most programs adopt the interface standards of the file access routines. This is not practical in all cases, notably for those programs where source code is not distributed or where the file interface is too intricate for easy conversion. In such cases, *GetData* can be used to convert data between formats used by different programs. The access routines for reading files are completely independent of the access routines for writing files, so format conversion can be accomplished by running *GetData* with read routines for one format and write routines for another.

This document is a manual for the *GetData* program, version 3.1. The document also includes specification of the interface standards for the file access routines and documentation of a specific set of access routines supporting several generally useful formats.

2 User's Guide to *GetData*

This section describes how to use the *GetData* program. Most of the description applies to any installation of *GetData*, but a few of the "niceties" are system dependent and may not be implemented in all installations. All such system-dependent features are mentioned in the text and are referenced under "system dependence" in the index. The manual will describe the system-dependent features as they are implemented on the ELXSI computer (ELXSI, San Jose, California, ref. 1).

The manual documents all limitations that arise from fixed array dimensions in the code (referenced under "limitations" in the index). All array limits were chosen liberally to accommodate most applications. The limits can be changed easily, in most cases by changing a single *parameter* statement in the code.

GetData is designed to be highly crash resistant. When it encounters an error, it terminates the failed command, prints an error message, and prompts for the next command. Such mundane errors as exceeding dimension limits or giving names of nonexistent files or signals are all detected. The only known internal program crashes involve data values too large for single-precision floating-point variables.

2.1 Running *GetData*

GetData is designed as an interactive program, that is, it reads commands from the user's terminal. The method of starting an interactive job is system dependent. On the ELXSI, you type

```
    GetData
```

The program will then display something like

```
    getData program
    time history data selection
    Richard Maine - NASA Dryden
    version 3.1.1 11 Sept 86
```

```
this run date: 21-Oct-86  time: 14:59:55

Help is available

getData:
```

The run date and time will be different, as might version number and date. The `getData:` prompt indicates that *GetData* is waiting for a command.

Some *GetData* runs can be slow or might need to wait for events such as tape mounting. The program can be run in a batch mode if desired. To run *GetData* in a batch mode, use an editor to prepare an input file containing all the *GetData* commands you would type to do the run interactively. Then specify this file as the system input file when you batch the *GetData* program. The methods of running a batch job and specifying its system input file are system dependent. On the ELXSI, the command to batch the *GetData* program with an input file called *commandFile* would be

```
batch 'commandFile> GetData'
```

All output that would have gone to the terminal in an interactive job will go to a system-dependent batch log file in a batch job.

The remainder of this manual is written as though *GetData* were being run interactively. It is implicit that the discussion also applies to batch runs if the references to the terminal are appropriately interpreted as applying to the system input file or batch log file.

2.2 Entering *GetData* Commands

Whenever you see the `getData:` prompt, the program is waiting for a command. This section describes general principles of command entry that apply to all *GetData* commands. The following sections describe the details of specific commands.

Input lines are limited to 256 characters including trailing blanks; some systems may enforce smaller limits.

If the last nonblank character in any line is an ampersand (`&`), it indicates that the command will be continued on the following input line. *GetData* will prompt with `more:` for you to enter the continuation line. Continuation lines can be further continued, subject only to a limit of 4096 characters in the concatenated command. Trailing blanks on the input lines are removed before concatenating the lines and therefore do not count towards this limit. The ampersand continuation characters are converted to blanks in the concatenated command; therefore, line breaks must occur at places where blanks are allowed. Concatenation of continuation lines occurs before any other processing of a command. Therefore, errors in a command will not be diagnosed until after the last line of the command, even if the error occurred on the first line.

If the first two characters of a command (except for possible leading blanks) are dashes (`--`), that command is considered to be a comment and is ignored. A completely blank command is also allowed as a comment. Comments are most useful in *do* files (section 2.11) and batch job inputs but are also allowed interactively. Note that comments can have continuation lines, just like all other commands.

Noncomment commands are divided into fields separated by various delimiter characters. The most generally tell what delimiters are expected between the various fields of the commands. Whenever the discussion does not explicitly state otherwise, a blank delimiter is expected.

GetData ignores any superfluous blanks between input fields, around delimiters, and at the beginning and end of commands. You can freely use such blanks. Blanks are not allowed within input fields, except for a few special cases where quoted fields with embedded blanks are allowed.

The first blank-delimited field in any noncomment command specifies what command is being invoked. The remaining fields are arguments to the command. The arguments are described in the sections about the specific commands. Many of the commands and arguments can be specified using abbreviations and synonyms.

There are three syntactic classes of arguments: positional, keyword, and switch. The command

```
read myFile fSkew=.02 +interpolate
```

illustrates all three classes. A positional argument simply consists of a value; the order of the positional arguments implicitly defines which arguments go with which parameters. The `myFile` in the above example is a positional argument; the program knows by its position as the first argument that this is a file name. A keyword argument consists of an argument name or keyword, followed by an equal sign delimiter, followed by a value. The `fSkew=.02` in the example is a keyword argument. The value is assigned to the parameter identified by the keyword. A switch argument consists of an argument name preceded by a `+` or `-` sign. Switch arguments assign the boolean values *true* or *false* to their parameters, with a `+` giving the value *true* and a `-` giving the value *false*. Switch parameters may have antonyms; setting the value of an antonym is equivalent to setting the original switch to the opposite value.

All *GetData* input is case insensitive on machines supporting upper- and lowercase. The commands can be entered in any mix of upper- and lowercase.

2.3 *Help* Command

The *help* command provides access to an online help facility. The current implementation is highly system dependent; it directly uses the ELXSI help facility. The *help* command may not be available in all implementations of *GetData* and may function differently in some implementations.

The basic function of the *help* command is to list a help file one screen at a time. To display a help file interactively, type `help` followed by the name of the help file as a positional argument. For instance,

```
help help
```

will display the help file for the *help* command itself. Typing `help` with no argument is equivalent to typing `help help`.

GetData has help files on commands, subroutines, and topics. To obtain a list of all the available help files, type one of

```
help commands
help subroutines
help topics
```

The responses should look something like

```
 getData:  help commands
Searching the index ...
copy [cmd] -- copy data from input to output file
method [cmd] -- define interpolation methods
read [cmd] -- specify input data file(s)
show/list [cmd] -- list signal names
signals [cmd] -- define signals to be written
skew [cmd] -- define input signal skews
write [cmd] -- specify output data file name
do [cmd] -- execute a command file
help [cmd] -- help command
quit [cmd] -- exit the program normally
sys [cmd] -- execute a system command without exiting program

 getData:  help subroutines
Searching the index ...
activateCF [sub] -- activate needed calculated functions
activateFilt [sub] -- activate needed filters
allocateCF [sub] -- locate signals for calculated functions
allocateFilt [sub] -- locate signals for filter
doCF [sub] -- evaluate calculated functions
doFilt [sub] -- evaluate filters
reMapFilt [sub] -- reMap filters to compressed locations
calculations [topic] -- calculated functions in getData

 getData:  help topics
Searching the index ...
calculations [topic] -- calculated functions in getData
cpuTime [topic] -- cpu time estimates for getData on ELXSI
version [topic] -- version 3.1 changes to getData
```

Appendix 3.6.2 contains listings of all the help files in *GetData.*

Some details of the supplied help files are specific to the installation of the program at NASA Ames Research Center, Dryden Flight Research Facility (Ames-Dryden). For instance, there are references to the installation of the program under a specific user name.

The ELXSI *help* command has several other features, such as keyword searches, as described in reference 1. These features do work in *GetData*, but they are more useful when there are hundreds or thousands of help files than when there are only a handful; therefore, this document will not give details.

2.4 Basic Operation

This section describes the simplest *GetData* runs. It shows how to copy selected signals and times from an input file to an output file. All other *GetData* runs are built on this basic structure.

Once the program is started, there are five steps required in any *GetData* run. These steps use the *read*, *signals*, *write*, *copy*, and *quit* commands, as in the following example command sequence:

```
read someFile
signals alpha, beta, p, q, r
write myFile
copy time = 10:0:0:0 - 11:0:0:0
quit
```

Some jobs may involve additional steps, but these five are always included.

The *read* command specifies the name of the input time history file. In this example, the input file is named *someFile*. In response to the *read* command, the program opens the specified file and determines what signals are available. The *read* command opens and prepares the file for subsequent data transfer, but does not immediately read any data from the file.

The *signals* command specifies what signals are to be written on the output file. In this example, the signals are selected by name. The signal names can be delimited by blanks or commas. An alternative form of the *signals* command is

```
signals +all
```

which selects all the signals currently available. The natural placement of the *signals* command is after the *read* command, which defines the list of available signals.

The *write* command specifies the name of the output time history file. In this example command sequence, the output file is named *myFile*. In response to the *write* command, the program opens a file with the specified name and defines the names of the signals on the file. The signal names must have been defined prior to the *write* command and should not be subsequently changed. The *write* command opens and prepares the file for subsequent data transfer, but does not immediately write any data to the file.

The *copy* command copies data from the input time history file to the output time history file. The input and output files must have previously been opened using the *read* and *write* commands. The *copy* command causes the actual data transfer to occur.

The *copy* command also specifies the time segments to be copied. The *time* argument gives the start and end times of the segment in hours, minutes, seconds, and milliseconds. If a time segment is specified, all eight time fields must be present, whether they are zero or not. The eight time fields can be delimited by blanks, commas, dashes, slashes, periods, or any mixture of these; no significance is attached to the delimiter. Note, therefore, that 10:00:00.5 would represent 5 msec past 10—not 0.5 sec. The equal sign shown in the example is optional.

You can omit the time specification from the *copy* command, using just

```
copy
```

In this case, a time interval of 0 to 24 hr is assumed; this usually causes all the times from the input file to be copied (unless the input file has times outside this range, which is not common but occasionally results from special conventions). The *copy* command will skip any requested times not present on the input file.

You can have multiple *copy* commands to specify multiple time intervals. For instance, the command sequence

```
read someFile
signals +all
write myFile
copy time = 10:0:0:0 - 10:0:5:0
copy time = 10:1:0:0 - 10:1:5:0
copy time = 10:2:0:0 - 10:2:5:0
quit
```

copies all the data from three 5-sec time intervals of *someFile* to *myFile*. When multiple time intervals are requested like this, the intervals should be nonoverlapping and in time sequential order. Otherwise the frames on the output file may be out of time sequential order, which causes problems with many programs (including *GetData*). *GetData* will print a warning message if this occurs.

The *quit* command closes all files and exits the *GetData* program. This is the normal way of terminating a *GetData* run. A system-dependent end-of-file from the terminal will also be interpreted as an implicit *quit* command, but the explicit *quit* command is less confusing.

2.5 Controlling the Output Frame Times

All time history data files manipulated by *GetData* are organized into frames, also called records. Each frame contains a time value, called the time tag or frame time, plus values of the signals at or near that time. For the moment, we assume that the signal values are exactly at the frame time; section 2.7 discusses the more complex situation.

In the simple examples of section 2.4, the output frame times were exactly the same as the input frame times in the requested time interval. This is the simplest means of defining the output frame times, but it is not adequate for all applications. *GetData* provides two methods of defining the output frame times, controlled by the *thin* and *dt* parameters in the *copy* command.

The first method is based on thinning the input frame times. It sets the first output frame time in an interval to equal the first input frame time in the interval; thereafter, the output frame times are equal to every *thin*th input frame time until the end of the requested time interval. You select this method by specifying the value of *thin* as a keyword argument to the *copy* command. For example,

```
copy time=10:0:0:0-11:0:0:0 thin=2
```

means to make output frame times for every second input frame time between 10 and 11 o'clock. Negative or zero values of *thin* are illegal. The value 1 makes an output frame time for every input frame time in the interval.

The thinning method makes no attempt to generate a constant sample rate in the output; it operates strictly and simply by thinning the input. For instance, if the input frame times were at 0, 1, 2, 3, 4, 6, 7, 8, 9, 10, ... msec after the interval start time, then thinning by 2 will produce output frames at 0, 2, 4, 7, 9, ... msec after the interval start time; the algorithm makes no attempt to compensate for the "missing" frame at 5 msec after the start.

The second method of defining the output frame times is to specify a constant time increment or sample interval between output frames. You select this method by specifying a nonzero value of the time increment *dt* as a keyword argument to the *copy* command. For example,

```
copy time=10:0:0:0-11:0:0:0 dt=.002
```

means to make output frames spaced exactly (to floating-point precision) 2 msec apart between 10 and 11 o'clock. The input frames can be at any sample rate or can even be sampled at irregular times. Negative values of *dt* are illegal; a zero value disables this method and is equivalent to specifying `thin=1`.

The implementation of the *dt* parameter makes one exception to the principle of constant output sample rate. If there is no input frame within 1 sec after a proposed output frame time, the output frames will be omitted until after the next available input frame time. Thus the program will skip large time dropouts in the input but interpolate through small ones. A message will be printed so that you will be aware of the omission. Signal skews (section 2.7) are not considered in this algorithm, so it may have problems if all the signals on a file are skewed by a second or more (but that is not the best way to specify uniform large skews anyway). The 1-sec criterion separating small dropouts from large ones is currently hardwired into the code, thus the program will not work well with input data rates less than 1 sample/sec.

This feature is intended to make it easy to work with files having several disjoint time segments of data. A command like

```
copy dt=.02
```

will make output frames at 50 samples/sec during the time segments present on the input file but will not waste huge amounts of file space filling out 24 hr of interpolated data at this rate. Without the special treatment for large time dropouts, you would have to determine what time intervals were on the input file and make a separate *copy* command for each interval to achieve this result.

The distinctions between the effects of the *thin* and *dt* parameters are critically important for some applications. The *dt* parameter gives a constant output sample rate, which is required for some analysis techniques. However, the output frame times resulting from the *dt* parameter will generally lie between input frame times, requiring some form of interpolation of the data (see section 2.8). The *thin* parameter avoids interpolation (unless you have time skews or multiple input files) but does not guarantee a constant output sample rate unless the input sample rate is constant. The *thin* parameter often uses substantially less computer time.

The keyword parameter *nTimes* can be used in conjunction with either the *thin* or the *dt* parameters. The *nTimes* parameter specifies a maximum number of output frames that will be written. It is most useful for debugging, when you want to look at only the first few frames of output. For instance, the command sequence

```
read inFile
signals +all
write outFile
copy nTimes=5
quit
```

copies the first five frames of *inFile* to *outFile* without requiring you to specify the times of those frames.

GetData expects its input files to be in time sequential order. Furthermore, there is a finite tolerance of 0.1 msec used in several places to avoid roundoff problems. *GetData* prints a warning message whenever the input times are out of order or spaced less than 0.1 msec apart. Such input frames may cause some of the input data to be discarded. One consequence of the 0.1-msec tolerance is that *GetData* does not work well with sample rates greater than about 10,000 samples/sec. This tolerance cannot currently be changed without recompiling the program.

2.6 Merging and Splicing Input Files

Many applications require that data from several input files be combined and written on a single output file. *GetData* provides two mechanisms for combining data from multiple input files; we refer to these mechanisms as merging and splicing.

Merging is the combination of data for the same time interval from multiple files. The different input files contain data for different signals. To merge data from multiple files, you must specify all the file names, separated by commas, in a single *read* command. The *read* command may have continuation lines but must not be split into separate *read* commands. For instance, the command

```
read /user/maine/firstLongFileName, &
     /user/maine/secondLongFilename, &
     /user/maine/thirdLongFileName
```

opens all three specified files and allows you to merge data from them. However, the sequence of commands

```
read /user/maine/firstLongFileName
read /user/maine/secondLongFilename
read /user/maine/thirdLongFileName
```

opens the specified files one at a time. Each *read* command closes all previously opened input files. Therefore, this sequence leaves only the data from the third file accessible.

GetData is limited to 10 simultaneously open input files. It is also limited to a total of 2000 total input signals, no more than 1000 of which can come from a single input file.

When multiple input files are open, the *signals* command automatically determines which input file contains the data for each input signal used. There is no outward difference in the usage of the *signals* command. Because the input signals are selected solely by name, there is no way to indicate your intent when the same signal name appears on two or more of the input files being merged. You can resolve this ambiguity by first copying one or more of the input files to temporary files with renamed signals; section 2.9 describes how to rename signals.

The input files being merged are not guaranteed to have the same frame times. Therefore it is important to consider the issue of interpolation (section 2.8). If you use the *thin* parameter of the *copy* command, the thinning is based on the frame times of the first file in the file list of the *read* command; the data for all other files are interpolated as specified by the *method* command.

Splicing is the construction of a single output signal that is taken from different sources for different time segments. You specify splicing by inserting other commands between *copy* commands. There are two major forms of splicing, distinguished by what kinds of commands are inserted.

One form of splicing is to change the input file or files between two *copy* commands. The sequence of commands

```
read file1
signals alpha, beta
write outFile
copy time = 9:0:0:0 - 9:0:30:0
```

```
     copy time = 9:5:0:0 - 9:5:10:0
     quit
```

makes an output file with a 30-sec segment of data from *file1*, followed by a 10-sec segment from *file2*. This example assumes that both *file1* and *file2* have signals named *alpha* and *beta*; because there is no ***signals*** command between the *copy* commands, the previously specified signal list remains in effect.

Another form of splicing is to change the signal list between two *copy* commands. The sequence of commands

```
     read inFile
     signals alpha, beta
     write outFile
     copy time = 0:0:0:0 - 9:29:59:999
     signals alpha, betaBackup
     copy time = 9:30:0:0 - 9:30:59:999
     signals alpha, beta
     copy time = 9:31:0:0 - 24:0:0:0
     quit
```

copies *alpha* and *beta* except for a 1-min segment where *betaBackup* is substituted for *beta* (perhaps the primary *beta* data were invalid during that segment). This kind of splicing should be done with caution because *GetData* has few means of verifying that your specifications make sense; for instance, you might have spliced a totally unrelated signal in place of *beta*. The signals appear on the output file in the same order as listed in the *signals* command. For most purposes, the order of the signals is irrelevant because you select signals by name rather than by position. However, when you insert a *signals* command between two *copy* commands, the splicing depends on the order because the signal names may be changed.

A single output file contains only one signal name per signal; there is no record of any name changes that may have occurred by splicing. Also, the number of signals on an output file cannot be changed by splicing. If you try to splice a time segment with fewer than the original number of signals, the remaining signals will contain unpredictable garbage for that time segment; if you try to splice in more than the original number of signals, the extras will be ignored. The names and number of signals on an output file are established when the *write* command is encountered. Any subsequent *signals* commands cause splicing. Because of the potential for undetected errors, *GetData* gives a prominent warning message whenever this kind of splicing is attempted.

GetData does not support multiple output files open at the same time. Each *write* command can name only a single output file. You can have multiple *write* commands in a job, but each one closes any previously open output file and opens a new one. The sequence of commands

```
     read inFile
     signals alpha, beta
     write outFile1
     copy time = 9:0:0:0 - 9:0:10:0
     write
     signals p, q, r
     write outFile2
     copy time = 9:0:0:0 - 9:0:10:0
       ..
```

copies a segment of *alpha* and *beta* data to *outFile1* and then copies *p*, *q*, and *r* data to *outFile2* for the same 10-sec segment. The *write* command with no arguments causes *outFile1* to be closed without opening another output file. This is not necessary, but if you omit that command in this example, *GetData* will print a warning message about splicing when it encounters a *signals* command while an output file is open. Because there are no *copy* commands between this warning and the subsequent *write* command, no splicing would actually occur.

All the operations discussed in this section can be mixed in the obvious ways. A single *GetData* run can involve merging, both kinds of splicing, and changes of the output file. For example, the command sequence

```
read inFile1
signals alpha, beta
write outFile
copy time = 9:0:0:0 - 9:0:9:999
read inFile2, inFile3
signals alpha, betaBackup
copy time = 9:0:10:0 - 9:0:20:0
quit
```

involves merging and both kinds of splicing.

2.7 Applying Time Skews

The data files manipulated by *GetData* have only one time tag associated with each frame of data. The data frame generally contains several data values, which are actually measured at slightly different times. The data files do not contain explicit time tags for each individual data value. There would be substantial overhead in saving an explicit time tag with each measurement. Instead, we assume that the times of the individual measurements can be implicitly deduced from the frame time tag.

The simplest approach to computing the times of the individual measurements in a frame is to assume that they all equal the frame time tag. This is often an adequate approximation; the error is usually less than the sample interval and rarely more than a few times the sample interval. Analysis programs almost invariably assume that all the data values in a frame have the same measurement times. Some applications are very sensitive to errors in this assumption, giving significantly erroneous results if there are time errors of more than a few milliseconds (or even less). Other applications for the same data can tolerate time errors as large as seconds.

GetData is a utility program rather than an end application. The timing accuracy requirements for *GetData* depend on the application of the data. If the application is insensitive to time errors on the order of the sample interval, *GetData* can use simple approximations for the measurement times. If the application needs precisely time-tagged data, *GetData* must treat the time tags with corresponding accuracy, accounting for the differences between the actual measurement times and the frame time tags.

The time skew of a measurement is defined as the actual measurement time minus the frame time tag. The simple algorithms described previously approximate the skew as zero. When this is not adequate, *GetData* can apply nonzero skews. *GetData* assumes that the skew for each signal is constant from frame to frame; more complicated situations can be addressed by special patches, which are not provided as part of the standard program.

Time skews can arise from many causes. Most instrumentation systems sample the measurements sequentially during the time interval; it is convenient to define the frame time tag to be the time at the begining of the interval, giving a skew that is dependent on the measurement sampling sequence. The physical instruments have dynamic response characteristics that can often be closely approximated as time lags; the data sampled at a given time represent the physical value for a slightly earlier time. Signal-conditioning filters also cause lags in the data.

The user must determine the total skew from these and other sources; *GetData* has no way to calculate what the skew should be. There are two ways to specify skew values to be used by *GetData*: the *skew* command and the *fSkew* parameter of the *read* command.

The *skew* command is the primary means for specifying skews in *GetData*. Any command name beginning with `skew` will be accepted as a synonym. The body of the *skew* command consists of any number of parameters in keyword syntax, separated by blanks or commas. The parameter names must be the names of input signals or filtered signals. You cannot apply skews directly to calculated signals or output signals, but you can apply skews to the input signals used in a calculation. The parameter values are the skews in seconds. The command

```
    skew alpha=.02, beta=.02 p=-.01
```

defines *alpha* and *beta* to have skews of 0.02 sec and p to have a skew of -0.01 sec. The skew for any parameter not specified in a *skew* command defaults to zero.

Every time a *read* command is encountered, *GetData* resets all signal skews to zero. Therefore, the *skew* command must follow the *read* command. Furthermore, if you are splicing data from two files, you must repeat the *skew* commands after the second *read* command if the same skews apply to both files.

There are significant performance penalties for processing signal skews, and these penalties become larger as the skew becomes larger. For input files that have no active signals with skews or linear interpolation (section 2.8), the program uses a special-case fast algorithm. As soon as an input file has a single active signal skew or linear interpolation, the special algorithm no longer applies for that file and the performance becomes substantially worse.

There are also limits to the magnitudes of signal skew that can be applied. These limits are functions of several factors and can be increased if needed (but this will cause further performance degradation). There will be a warning message if you exceed the limits.

A second means for specifying skews to be used by *GetData* is the *fSkew* parameter of the *read* command. This parameter gives a skew that is added to the frame times of a file to obtain "corrected" frame times. This corrected frame time is used in place of the raw frame time throughout *GetData*. If any skews are specified in *skew* commands, they apply in addition to the file skew.

You give the *fSkew* parameter in keyword syntax, with the skew value in seconds. Each file named in a *read* command has an independent *fSkew* parameter, which must follow the corresponding file name, delimited by a blank. The command

```
    read file1 fSkew=5.6, file2, file3 fSkew=-50
```

specifies a skew of 5.6 sec for *file1* and -50 sec for *file3*. The skew of *file2* is not specified and defaults to zero.

The file skew is similar to an equal skew applied to every signal in the file, but there are subtle differences. The file skew affects the computation of the output frame times discussed in section 2.5, but the signal skews do not. More importantly, the file skew has no limitations or performance implications.

The file skew may safely be several hours (perhaps to convert between G.m.t and local time). Individual signal skews cause severe performance degradation in *GetData* when they exceed a few times the sample interval. Therefore, if all the signals in a file have skews that are large relative to the sample interval, it is most efficient to specify a file skew near the mean of the skews.

2.8 Interpolating in Time

Each output frame from *GetData* has a single time tag and data values at or near that time. As discussed in section 2.7, most applications assume that the data values were measured precisely at the frame time, and some applications are very sensitive to errors in this assumption. *GetData* must be able to produce output files suitable for applications with precise timing requirements.

However, the raw data measurements are seldom conveniently available at precisely the required times. Section 2.7 discusses how *GetData* computes the precise times of the input signals, and section 2.5 discusses how it determines the output frame times. If there is only a single input file with no signal skews and if the output frame times are determined by thinning the input frame times, then all the input signals will be available at precisely the output frame times. In more general cases, the signals must be interpolated to the required output times.

Note that interpolation applies only to input (or filtered) signals—not to output signals. In many cases, each output signal corresponds to an input signal of the same name, making the distinction moot. When an output signal is a calculated function of several input signals, it should be fairly obvious that the input signals need to be interpolated to common times before the calculation can be done. However, when an output signal is just a renamed version of an input signal, it is easy to get confused.

GetData provides two interpolation methods: hold-last-value and linear interpolation. The hold-last-value interpolation uses substantially less computer time but is inadequate for many time-sensitive applications. Linear interpolation provides more accurate results for those applications needing them. Higher order interpolation algorithms are possible, but it is difficult to justify their use in the context of imperfectly measured time history data. Note that there are some signals that cannot be meaningfully interpolated with any method other than hold last value; for instance, a digital word may just be a bit pattern without a reasonable numeric interpretation.

You use the *method* command to specify the interpolation methods. Any command name beginning with `meth` will be accepted as a synonym. The syntax of the *method* command is very similar to that of the *skew* command. The body consists of any number of parameters in keyword syntax, separated by blanks or commas. The parameter names must be the names of input signals or filtered signals. The parameter values must be either `hold` (for hold-last-value interpolation) or `interpolate` (for linear interpolation); any values beginning with `h` or `i` will be accepted as synonyms. For example, the command

```
    meth alpha=interp beta=i alt=hold-last-value mach=h
```

specifies linear interpolation for *alpha* and *beta* and hold-last-value interpolation for *alt* and *mach*.

If most of or all the signals in an input file will use the same interpolation method, you can simplify the specification by using the *hold* or *interpolate* switches on the *read* command. These switches control the default interpolation method to be used for any signal in the input file not named in a *method* command. The *interpolate* switch is an antonym for *hold* and can be abbreviated to anything beginning with `interp`. Each file named in a *read* command has independent *hold* and *interpolate* switches, which must follow the corresponding file name, delimited by a blank. If neither switch is specified for a file,

```
read file1 -hold, file2, file3 +interp
method dw1=hold alpha=interp
```

specifies linear interpolation for signals on *file1* and *file3* (`-hold` is equivalent to `+interp`). The method for *file2* is not explicitly specified, so it defaults to hold-last-value interpolation. The signal *dw1* will use hold-last-value interpolation, and *alpha* will use linear interpolation, regardless of which input file they are on.

All previous interpolation method specifications are discarded whenever a *read* command is encountered. Therefore, the *method* command must follow the *read* command. Furthermore, if you are splicing data from two files, you must repeat the *method* command after the second *read* command if the same interpolation methods are to be used for both files.

Linear interpolation requires more computer time than hold-last-value interpolation. For input files that have no active signals with skews (section 2.7) or linear interpolation, the program uses a special-case fast algorithm. As soon as an input file has a single active signal skew or linear interpolation, the special algorithm no longer applies for that file and the performance becomes substantially worse.

2.9 Selecting and Defining Signals

The *signals* command defines the signals to be written on the output data file. Section 2.4 describes the simplest forms of the *signals* command, and section 2.6 discusses the role of the *signals* command in merging and splicing. We now document the syntactic details and full capabilities of the *signals* command. The most important feature not covered in previous sections is the ability to define signals as calculated functions.

The full syntax of the *signals* command is

`signals` [`+all`|`+add`|`+delete`] *outSig1*[=*expr1*] *outSig2*[=*expr2*] ...

where the square brackets ([ ]) indicate optional entries and the vertical bars (|) separate alternatives. Any command beginning with `sig` will be accepted as a synonym for the *signals* command.

The optional switches *all*, *add*, and *delete* specify what will be done with the remaining arguments. No more than one of these switches is allowed in a single *signals* command. If one of these switches is present, it must be the first argument of the command. Only the + form of the switches is recognized; you cannot, for instance, specify `-delete`.

If none of the optional switches is specified, the remaining arguments define the signals to be written; any previously defined output signal definitions are discarded. Sections 2.4 and 2.6 show examples of this usage of the *signals* command.

If the *add* switch is specified, the remaining arguments define additional signals to be written. These new signal definitions supplement, rather than replace, any previous output signal definitions. If there are no previous definitions, the effect is the same as if the *add* were omitted. The *add* switch is a convenience feature that allows you to break a long *signals* command into a sequence of shorter ones. For example, the command sequence

```
signals alpha beta
signals +add p q r
```

is equivalent to the single command

```
    signals alpha beta p q r
```

The *add* switch is often useful in conjunction with command sequences also involving the other *signals* command switches.

If the *all* switch is specified, the output signal list will be defined to consist of all currently available signals. This switch is heavily used; some applications would be unduly burdensome without it. If *all* is specified, any remaining arguments of the *signals* command are ignored. (Future versions of *GetData* might generate an error message if such discarded arguments are present.) Section 2.4 shows examples of the *all* switch.

If the *delete* switch is specified, the remaining arguments specify signals to be deleted from the output list. Anything beginning with **+del** will be accepted as a synonym. When this switch is used, the optional expressions in the signal definitions are irrelevant; only the names of the signals to be deleted are required. Any expressions present will be ignored. This switch is most useful in command sequences also involving the *all* switch of the *signals* command. For example, the sequence

```
    read inFile
    signals +all
    signals +del alpha beta
    write outFile
    copy
    quit
```

copies all the signals except *alpha* and *beta* from *inFile* to *outFile*. If *inFile* had many signals, any other way of specfying this operation would be laborious.

The remainder of the *signals* command is a list of output signal names and optional expressions. The signal names are separated by blanks or commas. If an expression is specified for a signal, the expression is separated from the signal name by an equal sign; there may be blanks on either side of the equal sign. The signal names are limited to 16 characters and cannot contain commas, equal signs, quotes (single or double), parentheses, or embedded or leading blanks. Subsequent usage of the data file will be easier if you also avoid plus and minus signs and if you start each signal name with a letter, but these suggestions are not enforced. The output signal names must not be quoted. Like all other *GetData* input, signal names are case insensitive.

The optional expressions define how the output signals are to be computed. If the expression for a signal is omitted, the default computation sets the output signal equal to an input (or calculated) signal of the same name. All the examples given previously used this default. Note that even though an output signal and an input signal may have the same name, *GetData* always considers them to be separate entities. For example, the command

```
    signals +delete alpha
```

deletes only the output signal definition for *alpha*; it does not affect the existence of an input signal named *alpha*.

Only simple linear expressions can be defined using the expressions in the *signals* command. If more complicated expressions are required, they must be coded in FORTRAN and installed as described in sections 3.1 and 3.2. If an expression is given, it consists of one to five terms in the forms

sign constant
sign signal
*sign constant * signal*

Embedded blanks are not allowed in expressions; the blanks in these form descriptions should not be included literally in the expressions. The *sign* is either + or -; it may be omitted from the first term of an expression. The *constant* is an unsigned real constant with no exponent part. The *signal* is the name of an input or calculated signal; this includes only calculated signals defined by calculated function subroutines (section 3.1) or filter subroutines (section 3.2), not calculations defined by expressions in the *signals* command. The signal names follow the same syntax rules as output signal names. The signal names in an expression may be enclosed in quotes (either single or double, but they must match). If a signal name in an expression contains plus or minus signs, or if it starts with a digit or a dot, then it must be enclosed in quotes to avoid possible misinterpretation. Each expression is limited to a length of 80 characters.

All signals used in the computation are skew corrected and interpolated to the output frame times as specified by the *method* command.

The simplest and most common use of expressions in the *signals* command is to define an output signal equal to an input signal of a different name. For example, the sequence

```
read inFile
signals +all
signals +delete pitch
signals +add q=pitch
write outFile
copy
quit
```

copies everything from *inFile* to *outFile*, renaming *pitch* to *q*. The following example defines an average elevator position (*de-avg*) and a corrected angle-of-attack (*alpha-cor*) signal.

```
signals de-avg=.5*"de-left"+.5*"de-right" &
        alpha-cor="alpha-raw"+3.125*q
```

Note the usage of quotes in the expressions but not in the output signal names. Other common kinds of expressions include sign corrections, as in

```
signals an=-an
```

and constants, as in

```
signals altitude=0
```

Constant signals might be used as placeholders for unavailable data.

The expression parser is quite crude; do not be confused by its similarity to FORTRAN syntax. It cannot handle any forms other than those listed. For instance, the multiplying constant must always precede the signal name instead of follow it. Exponent form (for example, 1.e-3) is not accepted for constants. Parentheses are not recognized. There can be no blanks in an expression, except around the equal sign.

The parser does not strictly enforce the rules for signal name syntax in all contexts. You can sometimes get by with expressions not meeting the stated syntax rules. For instance, 1*3 is interpreted as a constant 1 times a signal named *3*, even though the 3 is not enclosed in quotation marks. Such expressions are confusing and are not guaranteed to work with future parsers; they should be avoided.

The only optimization of the expressions is a special case for expressions consisting of a single signal name with the multiplying constant omitted or equal to +1.0. (The large majority of expressions have this form.) Expressions such as 2+2 will work, but the addition will be repeated at every time point, which is a horrible waste of computer time.

The signal name *1.0* is reserved for internal use. If you have an input signal with this name, references to that input signal will not give the correct results.

Ill-formed expressions will give an error message and substitute a blank expression, which will give the value 0. If any of the signals used in the computation of an expression is unavailable, an error message will be printed and the value 0 will be used for that signal. Although currently inactive, the expression will be remembered and may become valid after a subsequent *read* command.

The program does not currently detect the occurrence of multiple output signals of the same name, but files with such duplicate names may cause difficulties for you in the future. If there are multiple input signals with the same name, there is no way to specify which one you want; the result is not guaranteed to be repeatable.

2.10 Showing Signal Definitions

The *show* command shows information about the currently defined signals. The command `list` is accepted as a synonym. The current version of the *show* command has no arguments and can give voluminous output to the terminal if many signals are defined; future versions may include arguments to allow more selective display.

There are three sections of the display from the *show* command: input signals, calculated signals, and output signals. The input signals section shows the names of the signals available on all the currently open input files. If any filter subroutines are installed (section 3.2), this section also includes the names of the filtered signals.

The calculated signals section shows what calculated function subroutines are installed (section 3.1) and what signals they calculate. Parentheses around the name of a calculated signal indicate that the signal cannot be calculated because other signals required for the calculation are missing. The signals shown in parentheses are not counted as available; they are included in this display only to document which calculated signals are installed.

The output signals section shows the names and expressions of all currently defined output signals. If an output signal cannot be calculated because it depends on unavailable input signals, the output signal name will be shown in parentheses; the value 0 will be used for any such output signal.

2.11 Automating Command Sequences

Some *GetData* runs require more command input than is reasonable to enter interactively. The most [illegible] over a hundred names are not unusual. Interactive input of such long lists is difficult and error prone. The *do* command provides a means of automating such input.

The *do* command takes a single argument, which is the name of a file that we call a command file. The allowable format for file names is system dependent. The *do* command will cause *GetData* to begin reading command lines from the specified file. The file should be a normal text file containing *GetData* commands exactly as they would be typed interactively; it can include continuation lines and comments. The file can contain any number of *GetData* commands. Any *GetData* command, except for a nested *do* command, can appear in a command file.

After executing all the commands in the command file, *GetData* will again prompt for interactive commands from the terminal (unless the command file contained an explicit *quit* command, which is allowed). For consistency, *do* commands are also allowed in batch runs, although they are not as necessary in a batch context.

The *do* command is most useful when the same sequence of commands will be used in several *GetData* runs. The command sequence then need be entered only once into the command file. If a command is particularly long, it can be useful to put it in a command file even if only a single run is intended. Interactive typing of long commands is very error prone; putting long commands in a command file makes it less likely that they will be mistyped, and it makes correcting typing mistakes less painful.

For an example of a *do* command, suppose that the file *sigs* contains the lines

```
-- standard signal list for aero group
signals &
  alpha = aa1022 &
  beta = aa1023 &
  p = rg0002 &
  q = rg0003 &
  r = rg0004 &

       ⋮

  mach = cf0001 &
  alt = cf0002
```

with several dozen similar lines replacing the ellipsis. This is typical of many command files; it selects a fairly large number of signals and changes their names from forms meaningful to the instrumentation engineers to forms more meaningful to data analysts. The command sequence

```
read inFile
do sigs
signals +delete alpha beta
signals +add alpha=bb1022 beta=bb1023
write outFile
copy
quit
```

uses most of the signal definitions from the *sigs* file but substitutes different definitions for *alpha*

2.12 Running System Commands From *GetData*

There are many circumstances where, in the middle of a *GetData* run, you want to run some system command. For instance, you may want to see the list of files in a directory because you do not recall the exact name of a file you need. The *sys* command in *GetData* provides this capability.

This command is highly system dependent and may not be installed in all implementations of *GetData*. It may have some limitations in other installations.

Anything beginning with **sys** will be accepted as a synonym for the *sys* command. The remainder of the command after the command name is any legitimate system command, complete with any needed arguments. After the specified command completes, control will return to *GetData*. The ELXSI implementation will return to *GetData* even if the system command aborts for some reason; this may not be true of all implementations on other systems. Some system commands may cause strange effects too diverse to catalog here.

For example, on the ELXSI, the command

```
sys files
```

will list the names of the files in your current directory. The command

```
sys to monty "I'll get to it later.  I'm busy now."
```

sends a message to another user without exiting *GetData*. Finally, the command

```
sys emacs cmdFile
```

enters the *emacs* editor to create a file called *cmdFile*; you might do this to create a command file to be executed by the *do* command (section 2.11.)

2.13 Specifying File Formats

The file formats supported by *GetData* are determined by the particular set of time history file interface routines (section 3.3) installed in the program. The available file interface routines may vary substantially at different sites.

The *write* command of *GetData* has an optional third argument used to specify the format of the output data file. For example, the command

```
write outFile unc2
```

specifies that *outFile* is to be written in *unc2* format. If the third argument of the *write* command is omitted, it defaults to cmp2. (On systems that do not support this format, you might want to change the default to unc2.)

The precise interpretation of this argument depends on the time history file writing routines installed. Some specialized routines that support only a single file format may ignore the argument. The default routines currently installed at Ames-Dryden support multiple formats as specified by this argument. The formats currently supported by these routines are the following:

unc2 uncompressed 2 format—This is a binary uncompressed format appropriate for use by many computer programs.

cmp2 compressed 2 format—This is a binary compressed format for compact storage of large data files. This format uses machine-specific features and is not included in the portable version of the code. Similar formats could be implemented on other machines.

asc1 ASCII 1 format—This is an ASCII format intended primarily for tape transfer between different systems. The format is highly portable. Files in this format can also be displayed on a terminal screen or printed, although the *lis1* format is more legible. The ASCII format is far less efficient than binary formats, and it should be used only in circumstances where the binary formats are inadequate, notably transfer between incompatible machines. The *asc1* format consumes about five times the file size of *unc2* format and requires an order of magnitude more processor time.

lis1 list 1 format—This is an ASCII format suitable for printing or screen display. The format is intended only for human use; there are no routines provided for reading a file in this format.

Section 3.3 gives details of these formats.

There is no corresponding argument in the *read* command to specify the formats of the input data files. The time history file reading routines are normally expected to automatically determine the formats of the input files. The routines currently installed at Ames-Dryden automatically recognize and read the *unc2*, *cmp2*, and *asc1* formats. On some systems, this automatic format recognition may be difficult to implement. Furthermore, there may be file formats in use that are difficult to automatically distinguish on any system. In such cases, separate file reading routines will be required for each format. Should this prove burdensome, it would not be particularly difficult to add arguments to the *read* command to specify the input file formats. That would still require, however, that the user know which format is correct for each input file; it is far more convenient to determine the format automatically where feasible.

If the desired formats are supported by the routines installed in *GetData*, file format conversion is done as an automatic part of the copy operation. With the routines installed at Ames-Dryden, the sequence

```
read inFile
signals +all
write /dev/tape/001234 asc1
copy
quit
```

copies all the data from *inFile* to tape number 001234, writing the tape in ASCII 1 format. The file *InFile* may be in any supported format (except *lis1*, which is not supported for reading). The sequence

```
read /dev/tape/005678
signals +all
write outFile
copy
```

copies all the data from tape number 005678 to *outFile*, writing the file in the default compressed 2 format.

3 Programmer's Guide to *GetData*

This section documents the FORTRAN code of the *GetData* program. The emphasis of the documentation is on those areas of the code most likely to need modification for some purpose. Some portions of the code are system dependent and must be modified to install *GetData* on different computer systems.

In addition, there are several modules (sets of routines) intended to be user modifiable. It is possible, of course, for a user to modify any routine in *GetData*. The routines labeled as user modifiable are specifically designed for the installation of customized code. The interface to these modules is defined in such a way that the user can write customized versions without understanding the details of the rest of the program.

3.1 Calculated Function Modules

A calculated signal (also called a calculated function) is a signal that is computed as a function of other signals rather than being directly read from an input file. The ***signals*** command allows the definition of some simple calculated signals as described in section 2.9. Calculations more complicated than those supported in the ***signals*** command can be implemented in calculated function modules.

Up to five calculated function modules, called *CF1* to *CF5*, can be installed in *GetData*; this limit can be easily modified. Each module can define an arbitrary number of calculated signals (subject to the limit of 2000 total signals from all sources). Each calculated function module consists of three FORTRAN routines (plus any subroutines that the three primary routines might require). The basic *GetData* program includes empty routines for all five calculated function modules. To install a calculated function module, you must create a *GetData* program with the customized routines replacing the corresponding empty routines provided with the basic program. The procedure for doing this installation is system dependent.

The routines of the xth calculated function module are named *allocateCFx*, *activateCFx*, and *doCFx*; for example, the routines for the *CF1* module are *allocateCF1*, *activateCF1*, and *doCF1*. The general roles of these routines are as follows:

allocateCFx declares the names and allocates channel numbers for the signals calculated by this module. This routine also locates all the input signals required for the computations; it disables any calculations that cannot be done because of unavailable inputs. This routine is called before any calls to *activateCFx* or *doCFx*. It may be called multiple times in a single job if multiple read commands are executed. The routine must redo all allocations on each call.

activateCFx activates needed calculations. This routine determines which calculations are needed for the currently requested output signals. It activates those calculations and declares their input signals to be needed. This routine may be called multiple times as the list of requested output signals changes. It will always be called at least once between any call to *allocateCFx* and subsequent calls to *doCFx*.

doCFx evaluates calculated signals. This routine performs the actual computation of the calculated signals. It uses channel numbers from *allocateCFx* and activation flags from *activateCFx*. This routine is called one time for each output frame.

The detailed interface specifications for these routines are given in the help files (app. A). The sample

Each calculated function module can use signals from the input files, the filter module, and lower numbered calculated function modules. It cannot use signals from higher numbered calculated function modules or signals defined in the *signals* command. The calculations are performed immediately before writing each output record; they have no intrinsic sample rates. The input signals used in the calculations are all skewed and interpolated to the output times as specified by the *method* command before the calculations are performed.

The calculated function modules are intended primarily for calculations that give each output value as a function of input values at the same time. Slight extensions are possible; for instance, it is possible to implement a simple differentiator in a calculated function module by internally saving time and data values from the previous output frame. Such extensions are highly dependent on the output frame times. Computations that involve substantial interdependence of data in different frames are probably best done in a separate program.

Once a calculated function module is installed, the usage of the calculated signals is substantially the same as the usage of signals read from input files. For most purposes, the user need not even be aware of the distinction between calculated and input signals. The only major distinction is in determining which input files are required. The calculated signals will be available only if their required input signals are available. This list of required input signals should be documented for each calculated signal.

3.2 Filter Module

Filters cannot be conveniently implemented in the normal calculated function modules because digital filters are inherently linked to specific sample rates, whereas the normal calculated function modules do not have inherent sample rates and may be called at different rates, depending on the requested output. Therefore, *GetData* makes special provisions for a filter module. Only a single filter module is currently allowed; this module can support multiple filters.

The basic *GetData* program includes empty versions of the filter routines. To install a filter module, you must create a *GetData* program with the customized routines replacing the corresponding empty routines. The procedure for doing this installation is system dependent.

The fundamental difference between the filter module and the other calculated function modules is that the filter routines are linked to the input frame times instead of the output frame times. This allows the user to freely select output frame times without affecting the filter characteristics. The interface to the filter module provides for simultaneous independent filters on different input files; this complication does not arise in the other calculated function modules.

The filter module consists of the routines *allocateFilt*, *activateFilt*, *reMapFilt*, and *doFilt*. The general roles of these routines are as follows:

allocateFilt declares the names and allocates channel numbers for the filtered signals. This routine also locates the unfiltered signals used as inputs to the filter module. There may be multiple calls for the same input file number if there are multiple *read* commands; each call must completely redo the allocations for the specified input file number.

activateFilt activates needed filters. This routine determines which filtered signals are needed for the currently requested outputs. It activates those filters and declares their unfiltered input signals to be needed. There may be multiple calls for the same input file number as the list of requested output signals changes. *ActivateFilt* will always be called at least once between any call to *allocateFilt* and subsequent calls to *doFilt*.

reMapFilt remaps filter channel numbers to compressed locations. This routine remaps the channel numbers used by the filter subroutines. The channel numbers used in *allocateFilt* and *activateFilt* reserve channels for all signals available on each input file. For efficiency, the actual processing uses a data vector composed of only the signals needed, with the unused signals omitted. Subroutine *reMapFilt* remaps the channel numbers of all signals used in the filter module to channel numbers in this compressed data vector. *ReMapFilt* is called at least once between any call to *activateFilt* and subsequent calls to *doFilt.*

doFilt evaluates filtered signals. This routine performs the actual filter computations. It uses the channel numbers from *reMapFilt. DoFilt* is called one time for each input frame of each open input file.

The detailed interface specifications for these routines are given in the help files (app. A). The sample routines mentioned in the help files are listed in appendix B.

The filter interface conveniently allows only recursive causal filter forms; that is, the filters can depend only on prior and current data, not on future data. There is no easy way to run forward–backward filters or smoothers. Note that you can skew the filtered result to approximately compensate for the group delay of the filter.

The filters can use only signals that come directly from an input file. Calculated signals cannot be filtered (though they can use filtered inputs, which normally achieves about the same effect). The input signals used for the filters are raw, without skew corrections or interpolation. The filtered result may be skewed and interpolated in the same way as signals read from the input files. Normally, the appropriate skew for the filtered signal is different than that for the raw signal.

3.3 File Interface Modules

GetData uses the time history file interface modules for all operations on time history files. These modules are intended to be used in any program that reads or writes time history data files; the modules have no dependence on internal data structures of *GetData.* The read and write modules are independent to facilitate format conversion applications where a program uses the read module for one format and the write module for a different format.

To use *GetData* with a particular set of read and write modules, you must create a version of *GetData* with the customized routines replacing the standard ones. The procedure for doing this installation is system dependent.

The basic *GetData* program includes a set of read and write modules that simultaneously support multiple formats. Section 3.4 describes the specific formats supported by these modules. The read module automatically determines which of the supported formats to use for each input file; the supported formats were specifically designed to facilitate such automatic determination. It may not be practical to implement the automatic format determination on all systems. The write module requires explicit specification of which format to use for each file. Both the read and write modules are structured to allow easy addition of more formats. These multiple-format modules reduce the necessity for creating multiple versions of *GetData* with different read and write modules.

The file read module consists of seven routines: *openR*, *rSigs*, *sigsR*, *chansR*, *rewR*, *fSeek*, *fRead*, and *closeR*. The general roles of these routines are as follows:

openR opens a file for reading. The file name is supplied as an input argument. This routine must value of *true* if the open is successful. If the open fails for any reason, the function value returns *false.* The most common reason for an open failure is that the specified file does not exist; security limitations or unsupported file formats can also cause failures.

rSigs finds names of the available signals. This routine may be called any time after *openR* and before *closeR.* Use of this routine is optional.

sigsR specifies which signals are to be read. This routine selects signals by name. This routine can be called any time after *openR* and before *closeR.* The data vectors returned from subsequent calls to *fRead* and *fSeek* will contain the signals specified in the most recent call to *sigsR* or *chansR.* When a file is opened by *openR,* it is initialized to select all available signals; this default remains in effect until the first call to *sigsR* or *chansR.*

chansR specifies which channels are to be read. This routine is similar to *sigsR,* except that the signals are specified by channel number instead of by name. The use of *sigsR* is usually preferred. *ChansR* is provided primarily for support of older programs and may eventually be phased out.

rewR positions the file at the first frame. This routine repositions the file so that subsequent calls to *fRead* will return data starting at the first frame of the file. The file is automatically positioned to the first frame by *openR,* so an initial call to *rewR* is not needed.

fSeek positions the file to a user-specified time and returns the first frame of data after that time. The routine returns a function value of *true* if the operation is successful. If there are no data after the specified time, the routine returns a function value of *false*, and the returned data vector is undefined.

fRead returns the next sequential frame of data on the file. The routine returns a function value of *true* if its operation is successful. If there are no more data on the file, the routine returns a function value of *false*, and the returned data vector is undefined.

closeR closes the file. This routine should be used to close any file opened by *openR.* After *closeR* is called, no further reference can be made to the file unless it is subsequently reopened.

The detailed interface specifications for these routines are given in the help files (app. A).

The file write module is somewhat simpler than the file read module because there are no issues of file positioning or signal selection. You must write the frames in time-sequential order, and you must provide values for every signal in every frame. The file write module consists of three routines: *openW*, *fWrite*, and *closeW*. The general roles of these routines are as follows:

openW opens a file for writing. The file name is supplied as an input argument. Other input arguments specify the names of the signals and the file format. The interpretation of the file format argument may vary in different implementations of the module; some implementations may ignore it. This routine must be called before any other reference to a file by the write module.

The routine returns a function value of *true* if the open is successful. If the open fails for any reason, the function value returns *false.* Common reasons for open failures include invalid file names, security limitations, and unsupported file formats.

fWrite writes a single frame of data to a file opened by *openW.* It should be called once for each frame to be written. The frame times must be in time-sequential order; this is not currently enforced but may be in future versions.

closeW closes the file. This routine should be used to close any file opened by *openW*. After *closeW* is called, no further reference can be made to the file unless it is subsequently reopened. The *closeW* call is mandatory; the created file is not guaranteed to be readable if it is not closed with *closeW*.

The detailed interface specifications for these routines are given in the help files (app. A).

Time history data files can be accessed either through the time history data file read and write modules or through normal FORTRAN input–output statements. The same file can be accessed in different ways at different times. However, the two forms of access should not be mixed during a single open. If a file is opened with *openR* or *openW*, it should be accessed only through the file interface modules until it is closed. Any other reference to the file, even something as simple as a rewind, may disrupt the operation of the interface module.

The following sample program fragment illustrates the use of both the file read and file write modules. This fragment copies the signals named *alpha* and *beta* from *inFile* to *outFile*.

```
      external openR,sigsR,fRead,openW
      logical openR,fRead,openW
      integer nSigs,nAvail
      parameter (nSigs=2)
      double precision time,data(nSigs)
      character sigs(nSigs)*16
      data sigs/'alpha','beta'/

      if (.not.openR(11,'inFile',nAvail)) call abort('no inFile')
      call sigsR(11,sigs,nSigs)
      if (.not.openW(12,'outFile',nSigs,sigs,'unc2'))
     x  call abort('cant open outFile')
  100 if (fRead(11,time,data)) then
        call fWrite(12,time,data)
        goto 100
      endif
      call closeR(11)
      call closeW(12)
```

The subroutine *abort* referenced in this sample fragment is assumed to print an error message and stop.

3.4 File Formats

The file read and write modules included with the basic *GetData* program support the following four file formats:

unc2 uncompressed 2 format—This is an uncompressed binary format suitable for most internal and interprogram files.

cmp2 compressed 2 format—This is a compressed binary format intended to save space when used for large files. It is substantially more complicated than *unc2* format. The implementation of this format is highly system dependent, so the format is disabled in versions of the code intended

asc1 ASCII 1 format—This format is intended for tape transfer between different systems. The format is highly portable. This format is very inefficient, both in file size and processing requirements, so it should not be used when one of the binary formats will work. Files in this format can be listed on terminal screens or printed, but the result is not particularly easy to read.

lis1 list 1 format—This format is intended for listing to terminal screens or printing. The read module does not support this format; files written in this format are only for human examination—not for input to computer programs.

Detailed descriptions of these formats are given in the help files (app. A). There is also a help file for *unc1* format, which is supported in some versions of the modules.

The following is a listing of a short sample file in ASCII 1 format:

```
format  asc 1
nChans        12
names          alpha          q              v               theta
an             ax             qbar           de              beta
p              r              phi
data001
 34988.023            3.8622436523438    -.27187347412109      357.82812500000
 5.2135009765625      1.0374450683594     .64163208007813E-01 152.17187500000
 .16679763793945     -.57655334472656E-01 .25144958496094     -.12891769409180
-2.9539794921875
 34988.049            4.0593261718750    -.17108917236328      357.92187500000
 5.2135009765625      1.0139465332031     .62057495117188E-01 152.25000000000
 .23110304027796E-04 .13263320922852      .44789886474609     -.42504119873047
-2.9539794921875
 34988.073            3.9017333984375    -.70299148559570E-01 357.92187500000
 5.2135009765625      1.0021972656250     .28367996215820E-01 152.25000000000
 .16679763793945      .13263320922852    -.43224334716797E-01-.42504119873047
-2.9539794921875
 34988.099            3.8623046875000    -.37266540527344      357.92187500000
 5.2135009765625      1.0256958007813     .81008911132813E-01 152.25000000000
 .33340454101563      .89379882812500     .64434814453125     -.12891769409180
-2.9539794921875
```

The same data in list 1 format look like

```
              alpha       q           v           theta       an
              ax          qbar        de          beta        p
              r           phi
09.43.08.023   3.8622     -.27187      357.83      5.2135      1.0374
               .64163E-01  152.17      .16680     -.57655E-01  .25145
              -.12892     -2.9540
09.43.08.049   4.0593     -.17109      357.92      5.2135      1.0139
               .62057E-01  152.25      .23110E-04  .13263      .44790
              -.42504     -2.9540
09.43.08.073   3.9017     -.70299E-01  357.92      5.2135      1.0022
               .28368E-01  152.25      .16680      .13263     -.43224E-01
              -.42504     -2.9540
09.43.08.099   3.8623     -.37267      357.92      5.2135      1.0257
               .81009E-01  152.25      .33340      .89380      .64435
              -.12892     -2.9540
```

The [illegible], unc2, and cmp2 formats are not printable

3.5 System Dependencies

GetData is coded in FORTRAN 77. It largely conforms to ANSI standard FORTRAN (ref. 2); this section describes all nonstandard or nonportable usages in *GetData.*

The program requires a full language FORTRAN 77 compiler; it makes extensive use of features included in the full language standard but not in the subset language. The items discussed in this section involve either extensions to the full language, details left unspecified by the standard, or system routines supplied independently of FORTRAN.

The program code is divided into several separate files. The discussions in this section are organized by the source code files to which they apply. The following items apply throughout the program.

***include* syntax**—The code is maintained on the ELXSI with common blocks and some other code fragments segregated into separate files. *Include* directives specify where these fragments should be inserted into the source code. Although most systems provide such a capability, the specific syntax varies widely. Some distributed copies of the code have the fragments already inserted, which simplifies the initial conversion issues but complicates subsequent program maintenance.

Precision—All floating-point variables in *GetData* are declared *double precision*, which is appropriate for scientific data on 32-bit systems. Double precision will work on 60-bit and 64-bit systems, but it is probably wasteful.

The current code does use single precision in two routines: *fRunc2* and *fWunc2*. These routines read and write *unc2* format records, which are defined to contain single-precision data values to conserve file space. The *fRunc2* and *fWunc2* routines appropriately translate between the double-precision data in their arguments and the single-precision data in the files.

GetData adheres to coding practices that facilitate easy changes of the precision. The program can be converted from double precision to single precision simply by replacing all occurrences of `double precision` with `real`. The only additional change required is in routine *oWunc1*, which deduces the number of signals in an obsolete *unc1* format file; a single-precision version should subtract only one word for the time variable instead of two.

Long names—*GetData* does not conform to the ANSI limit of 6-character symbolic names. No names longer than 15 characters are used except for a few ELXSI intrinsic names that are not portable anyway.

ASCII character set—*GetData* uses the full printable ASCII character set, including lowercase and special characters. Character comparisons are explicitly coded to be case insensitive. There is no explicit dependence on the ASCII collating sequence. The special characters are used only in noncritical places such as help file text; any legal characters can be substituted without impairing program functionality. No nonprintable characters are used.

Conversion to other character sets is a simple automatic translation. After such translation, the code should work in other character sets, including EBCDIC and uppercase-only sets.

Unit numbers—ANSI does not completely specify the set of allowable file unit numbers. File unit numbers in *GetData* are all specified by parameter statements to facilitate changing them. The unit numbers most likely to need changing are those for the standard input and output files (the terminal for interactive jobs); these are specified by the parameters *input* and *ouput* at the beginning of every routine. The code provided uses unit number 5 for *input* and 6 for *output.*

***open* statements**—*Open* statements are common places for system-dependent code. Several specific instances are mentioned later in this section. You may find that efficiency or operating convenience can be improved by making other changes to *open* statements, even if the program works as delivered.

The allowable file names for *open* statements are not specified by the standard and may be different on different systems.

***readOnly* parameter**—A nonstandard *readOnly* parameter is used in several *open* statements. This parameter reduces the chances of accidental file corruption and facilitates concurrent access to the same file by multiple jobs. It is currently used in routine *doDo* and in the *openR* routines for various file formats.

The *readOnly* parameter is noncritical and can be removed for systems that do not support it or an equivalent.

implicit none—The *GetData* code uses the *implicit none* statement. This statement helps detect coding errors but has no effect on the generated code. These statements can be removed safely if your system does not support this feature.

Initialization—*GetData* does strictly adhere to the standard in avoiding references to undefined variables. Any local variables that are required to retain their values between calls to a routine are declared in *save* statements. The program will therefore work correctly with compilers that allocate local variable storage on a stack.

The following items apply to the *rem.f* file. This file contains general utility routines used in *GetData* and several other programs.

***booboo* routine**—Subroutine *booboo* calls the ELXSI intrinsics *S$Init*, *DCI$StackTrace*, and *$Put* to format and print a traceback. This should be converted to calls to appropriate system routines on other systems. On systems that do not provide user-callable traceback routines, it may be possible to obtain a traceback by intentionally causing a run-time error. If there is no easy way to obtain a traceback, *booboo* can simply stop after printing its error message. The traceback is not critical except as a debugging aid. The program is not supposed to call *booboo* except as a last resort when a program bug or other unrecoverable problem is detected.

Subroutine *booboo* should never return to the routine that calls it. A call to *booboo* generally indicates that the program is not in a state to proceed successfully. A return may have unpredictable results, such as exceeding array limits.

***clock* routine**—Subroutine *clock* calls system routines *date* and *time* to obtain the date and time as printable strings. This function is used only for labeling. If corresponding routines are not available, you can change this routine to return blank strings.

String functions—Subroutines *upCase* and *loCase* and function *strEq* call ELXSI intrinsics for string case conversion and case-insensitive comparisons. Machine-independent versions are included as comments in the code. The machine-independent versions are substantially (up to an order of magnitude) slower than the ELXSI intrinsics. The machine-independent *upCase* and *loCase* make assumptions about the character set that are technically nonstandard but work in most environments, including EBCDIC and uppercase-only systems. (These routines leave the data unchanged on uppercase-only systems.)

***recLen* routine**—The *recLen* function uses a system-dependent error code. This function is used only for support of the obsolete *unc1* file format; it can be ignored if support for that format is not needed. If support for *unc1* format is needed, it is easy to deduce the required error code.

***sysErr* routine**—Subroutine *sysErr* calls the ELXSI intrinsic *$ErrorMsg* to print a detailed error message about the preceeding input–output error. This is used in *GetData* to help the user determine why a file could not be opened. Some systems may need the value returned by the *iostat* parameter of the *open* command. Therefore, this value is passed as an argument to *sysErr* even though the ELXSI does not use it.

This function is noncritical and can be deleted if no corresponding system error message features are available.

The following items apply to the *getCmd.f* file. This file contains the "front end" routines to read user commands from the terminal or alternative input files. It also contains code to implement some commands, such as *help*, that are pertinent to many interactive programs.

File kind intrinsics—The ELXSI intrinsics *FS$ReturnFileKind* and *FR$FileDescriptor* and the parameter *FS$TerminalFileKind* are used by the function *inCmd.* These intrinsics are used to determine whether input is coming from the terminal or not. This is then used to control whether the program echos the input. This allows the program to echo alternative input files to the terminal without duplicating the normal operating system echo of terminal input.

The echo function is noncritical. If your system cannot easily determine the type of input file, you can simply hard-wire the *echo* variable to *false.*

***doSys* routine**—Subroutine *doSys* is completely system dependent. The purpose of this subroutine is to execute a system command line from within the program.

Although very convenient, the *doSys* subroutine is not critical to the basic program operation. If you cannot implement an equivalent operation, you can make this subroutine print an error message saying that it is unimplemented. In this case, the *sys* and *help* commands will not work.

***doHelp* routine**—the *help* command is implemented in subroutine *doHelp* by using the *doSys* routine to access the ELXSI help utility. It therefore depends on both a working *doSys* subroutine and a compatible help utility. It is very unlikely that a fully compatible help utility will be available on anything but an ELXSI. Also, the *GetData* main program calls the *setHlp* entry with a specific directory name applicable only to the Ames-Dryden system.

Although useful, the *help* command is noncritical. You can run the program with *doHelp* changed to simply print an error message and return. If some other system help utility is available, you might reformat the help files as required and change *doHelp* to invoke the corresponding help utility. Alternatively, you could write code to perform at least the simplest function of the help utility. At its simplest level (a good one does much more), a help utility just opens a text file with a name constructed from the help argument and lists this file to the terminal screen. This would not be overly difficult to do in standard FORTRAN.

The following items apply to the file interface modules:

Delete in *openW*—The *openW* function should attempt to delete a file (if it exists) prior to writing a new file of the same name. This avoids potential problems if some characteristics (such as of *openW* uses the ELXSI intrinsic *FS$Delete* for this purpose. This code is "commented out" of the version intended for other machines.

The issue of incompatible file characteristics may not exist on some systems. In that case, or if operational use patterns assure that the old characteristics will always be compatible with the new usage, you can omit this call. Otherwise, you must substitute equivalent system-dependent code or use procedures that handle the problem external to the code.

Direct access open—Routines *oRasc1*, *oWunc2*, and *oWasc1* do direct access opens on files later used with sequential input–output statements. This is for ELXSI-specific performance and convenience reasons. There is no reason why standard sequential access opens should not work.

Block size—The *open* statements in *oRasc1* and *oWasc1* have block size specifications. These can be removed safely for most applications.

Rapid file positioning—Functions *fixedLen*, *nRecs*, and *fSec* have system-dependent code to do rapid file positioning. This is primarily of concern for very large data files. If this code cannot be converted easily, then *fixedLen* should be changed to always return the value *false*; this will prevent *fSec* and *nRecs* from ever being called.

Open in *openR*—The multiple-format version of the *openR* function needs to be able to read at least part of the first record of any supported file format without knowing which format the file uses. The supplied version does formatted input from the file, which may have been written with formatted or unformatted writes, depending on the file format. The standard does not preclude formatted input from files written with unformatted output, but it does not require all systems to support such an operation.

On some systems, this may be difficult to achieve in full generality. This may restrict the utility of the automatic format recognition on such systems, possibly requiring manual external specification of file characteristics.

***cmp2* format**—The *cmp2* file format is highly ELXSI specific. This format is disabled on versions of the code meant for porting to other systems. Similar compression ideas apply to many systems, but efficient implementation may require substantial work.

The following items apply to the *GetData* main program or the routines specific to *GetData*:

Command-line processing—The *GetData* main program calls the ELXSI *$CheckArgs* intrinsic to process the command line used to invoke the program. *GetData* does not actually use any command-line parameters; the only effect of this call is to check for an erroneous command line and give a reasonable error message. Without this call, some command-line errors may cause subsequent error messages in less obvious contexts. In most conversions, you should simply omit this call.

Performance testing code—The function *copyI* calls the ELXSI intrinsics *OS$ReadCpuTimer* and *$SwitchVar*. Some associated variables are declared with the nonstandard type *integer*8*. These calls and variables are used to compute performance statistics if activated by a shell switch variable. The calls and associated code can be removed safely; they are primarily used for developmental testing and might not be present in versions distributed for production use.

Default format—Subroutine *pWrite* defines the default output file format to be *cmp2*. The *cmp2* format is not supported in the portable version of the file interface module; therefore, you will probably want to change this default to *unc2*.

3.6 Specific Conversions

This section documents the specific changes we have found necessary to convert *GetData* to some other systems. These conversions were done as part of the program validation, not for operational use. Therefore, we did not spend much time on obtaining the best efficiency or converting noncritical features. Furthermore, our experience on these systems is limited; there may be better ways to do some of the functions. For production versions, you will probably want to do further work, but these tested changes give a reasonable starting point.

This section just briefly lists the changes required for these conversions. Section 3.5 further discusses the conversion issues.

3.6.1 VAX–VMS Conversion— This section describes conversion of the code to run on a DEC VAX running VMS (Digital Equipment Corporation, Maynard, Massachusetts).

The following change was made throughout the code:

***include* syntax**—The syntax of all *include* statements was changed to

```
include 'whatever.com'
```

as accepted by the VAX compiler.

The following changes were made in the *rem.f* file:

***booboo* routine**—The calls to ELXSI intrinsics were removed from the code in *booboo.* We do not know whether appropriate substitutes exist.

String functions—The routines *loCase*, *upCase*, and *strEq* were changed to use the "commented out" machine-independent versions of the code. The VMS *STR$UPCASE* procedure could be used in *upCase*, but we do not know of corresponding VMS procedures for *loCase* and *strEq*.

***sysErr* routine**—Subroutine *sysErr* was changed to print just the error number. Obtaining a more reasonable error message seems to be quite complex.

The following changes were made in the *getCmd.f* file:

File kind intrinsics—The calls in *inCmd* to ELXSI intrinsics were removed and *echo* was hardwired to *false.* We do not know whether equivalent capability is easily accessible under VMS.

***doSys* routine**—The body of *doSys* was replaced with the single line

```
call LIB$Spawn(tail)
```

***doHelp* routine**—The body of *doHelp* was removed, and code to print a warning message was substituted.

The following changes were made in the file interface modules:

Direct access opens—The direct access and record length specifications were removed from the *open*

Block size specifications—The block size specifications were removed from the *open* statements in *oRasc1* and *oWasc1*.

Rapid file positioning—The calls to *fixedLen* and *fSsec* from *fSunc2* were removed, making it always call *fSgen*; *fixedLen*, *fSsec*, and *nRecs* were eliminated.

The following changes were made in the *GetData* main program and the routines specific to *GetData*:

Command line processing—The call to *$CheckArgs* in the *GetData* main program was removed.

Performance testing code—The calls to *OS$ReadCpuTimer* and *$SwitchVar* were removed from *copyI*. The associated variables and code were also removed.

Default format—The default format was changed from *cmp2* to *unc2* in routine *pWrite*.

3.6.2 UNIX Conversion— This section describes conversion of the code to run on an IRIS (Silicon Graphics, Mountain View, California) workstation using the *f77* compiler running under ATT UNIX Version V (AT&T Bell Laboratories, New York). Most of the conversion should also apply to other UNIX systems. In a few places, the converted code calls UNIX system functions. The details of how to call UNIX system functions from FORTRAN vary from system to system. None of the system calls are critical to the basic function of *GetData*; they can be omitted if they are hard to call on your system.

The following changes were made throughout the code:

***include* syntax**—The syntax of all *include* statements was changed to

```
$       include whatever.com
```

as accepted by the UNIX compiler.

Unit numbers—The unit numbers for terminal input and output were changed to 0.

implicit none—All *implicit none* statements were removed.

***readOnly* parameter**—The *readOnly* parameter was removed from several *open* statements. We know of no equivalent substitute.

The following changes were made in the *rem.f* file:

***booboo* routine**—The calls to ELXSI intrinsics were removed from the code in *booboo*. We do not know of any appropriate substitutes.

***clock* routine**—The calls to *date* and *time* were removed from the *clock* subroutine. Blank strings are returned. UNIX does provide functions to get appropriate strings, but it was too much work to figure out how to use them from FORTRAN on the IRIS.

String functions—The "commented out" machine-independent versions of the code were used in routines *loCase*, *upCase*, and *strEq*. UNIX provides system calls for the *upCase* and *loCase* functions, but the complications of using them from FORTRAN probably make them less efficient than the machine-independent versions.

***sysErr* routine**—Subroutine *sysErr* was changed to call the *perror* system function.

The following changes were made in the *getCmd.f* file:

File kind intrinsics—The calls in *inCmd* to ELXSI intrinsics were removed, and *echo* was hardwired to *false*. Equivalent capability probably exists under UNIX, but we did not investigate it.

***doSys* routine**—The body of *doSys* was replaced with a call to the UNIX *System* function.

***doHelp* routine**—The body of the *doHelp* routine was removed, and code to print a warning message was substituted.

The following changes were made in the file interface modules:

Direct access opens—The direct access and record length specifications were removed from the *open* statements in *oRascl*, *oWascl*, and *oWunc2*.

Block size specifications—The block size specifications were removed from the *open* statements in *oRascl* and *oWascl*.

Rapid file positioning—The calls to *fixedLen* and *fSsec* were removed from *fSunc2*, making it always call *fSgen*; *fixedLen*, *fSsec*, and *nRecs* were completely eliminated.

The following changes were made in the *GetData* main program and the routines specific to *GetData*:

Command line processing—The call to *$CheckArgs* in the *GetData* main program was removed.

Performance testing code—The calls to *OS$ReadCpuTimer* and *$SwitchVar* were removed from *copyl*. The associated variables and code were also removed.

Default format— The default format was changed from *cmp2* to *unc2* in routine *pWrite*.

Appendix A—Help Files

This appendix consists of listings of the help files installed on the Ames-Dryden Elxsi computer. Some of the details are specific to this installation of the program and would not apply to other sites. For instance, there are numerous references to specific file path names in */user/maine*. The sample subroutines mentioned in some of these help files are listed in appendix B. The samples of file formats are listed in section 3.4.

A.1 Program Help File

```
getData [cmd] -- select time history data times and signals

USAGE
  [/user/maine/commands/]getData

DESCRIPTION
  This program selects signals and time intervals from time history
  data files.  It can also be used to copy time history data files to
  different file formats.  The program is designed for interactive
  use.

  The program can apply time skews, interpolating data to the output
  times using either linear interpolation or hold-last-value
  algorithms.  Input can be merged from multiple asynchronous files.
  There is also provision for calculated parameters, defined by
  user-supplied subroutines.  Calculations consisting of simple
  linear combinations of signals can be defined interactively without
  writing Fortran code.

  The program resides in the directory /user/maine/getData/commands,
  with an alias in /user/maine/commands.  The useMaine command
  facilitates access to getData.

  There is a full internal help facility, which covers the commands
  within getData.

EXAMPLES
  /user/maine/commands/useMaine
  getData
    read infile
    signals &
      alpha=alphaf beta=betaf &
      p=x12345
    write outfile unc2
    copy times 7:30:15:000 - 7:31:0:000 dt=.1
    quit
  getData
    read infile
    signals +all
    write outfile
    copy
    quit
```

CAVEATS

The order of the read, signals, write, and copy commands is important. They should be in this order.

ERROR HANDLING

The program attempts to recover from all errors. Such mundane errors as exceeding dimension limits, or giving names of non-existant files or signals are all caught. The program should not crash, regardless of what junk you feed it for commands. Infinite or NaN quantities in the data may crash it. If you succeed in crashing the program in any other way, please let me know.

SEE ALSO

bindGetData, fileInterface, unc1, unc2, cmp2, useMaine, internal help

IMPLEMENTATION

Fortran program.

The time history data file interface routines are used to read and write the data files. See the help topic fileInterface for discussions of the file interface subroutines. You must write customized versions of these routines to use getData on data files not supported by the standard ones.

KEYWORDS

GetData, select signals/intervals for time history data files, time skews/interpolation/thinning/(sample rates/intervals)

AUTHOR Richard Maine - NASA Dryden
VERSION 3.3.1
DATE 3 Sept 86

A.2 Command Help Files

A.2.1 *Copy* Command—

```
copy [cmd] -- copy data from input to output file

USAGE
  copy
       time[s] = hh:mm:ss:mmm - hh:mm:ss:mmm
       dt=<dt>   thin=<thin>   nTimes=<nTimes>

PARAMETERS

  time
    Time interval to be copied. The default time interval is 0-24
    hours. If time is specified, all 8 time fields are required,
    even if they are 0. The time fields can be delimited by blanks,
    colons, dashes, slashes, or periods. Note that the last field of
    time is milliseconds, rather than decimal seconds, regardless of
    the delimiter used; thus 12:00:00.5 represents 5 milliseconds
    past noon - not half a second.

  dt
    Output sample interval. If a non-zero dt is specified, the
    output times will be at intervals of exactly (to floatting point
    precision) dt. If all input files drop out for a period of
    longer than 1 second, the corresponding times will be dropped out
    of the output file and a message will be printed. If dt is 0 or
    is unspecified, the output times will be determined by the thin
    parameter. It is illegal to specify both dt and thin.

  thin
    Thinning factor for output. If thin is specified, the output
    times will exactly equal the input times of the first file
    specified on the most recent read command, thinned by the
    specified factor. The input file skew is included in this
    calculation. The default for thin is 1, which results in no
    thinning. It is illegal to specify both dt and thin.

  nTimes
    Maximum number of time points to write. If nTimes is non-zero,
    the copy operation will stop after that number of output times
    are written (or at the requested end time, whichever comes
    first). This is an easy way to look at the first few time points
    on a file. Anything beginning with nt is accepted as an
    abbreviation. If nTimes is 0, it is ignored. The default is 0.
```

DESCRIPTION

This command requests that a time interval be copied. The input and output data files, and the signals must have been previously specified. For multiple time intervals, use multiple copy commands.

All input data will be interpolated to the output times using either linear interpolation or hold-last-value as specified by the methods command.

EXAMPLES

```
copy
copy times 7:30:15:000 - 7:31:0:000 dt=.1
copy times=7 30 15 0 7 31 0 0  thin=2
copy nTimes=5
```

CAVEATS

For most applications, the time segments should be in order of increasing time and should not overlap. Many programs can not deal well with files having unordered times. Future versions of getData may disallow writing such files.

ERROR HANDLING

There will be an error message whenever the times from an input file are out of order or when out-of-order times are written to the output file.

SEE ALSO

read, signals, write, show, skew, methods

KEYWORDS

copy command,
copy data,
set/specify/select time/(sample interval/rate)/dt/
(thinning factor)/nTimes

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.2
DATE 17 Nov 86

A.2.2 *Do* Command—

```
do [cmd] -- execute a command file

USAGE
  do <command_file_name>

DESCRIPTION
  The do command causes the program to begin taking command lines
  from a text file.  The file can contain any command that could be
  entered from the terminal, except for nested do commands (which are
  disallowed to avoid possible recursion problems).  Following
  execution of all commands in the file, control returns to the
  terminal (unless the command file included a quit or other command
  that terminates the program).  Command files are appropriate for
  regularly-used long command lines or series of command lines.

  Note that only the actual command lines are obtained from the
  command file.  Any other input required for the commands is still
  obtained from the terminal.

  The following details of command line entry also apply to commands
  entered directly from the terminal, but are particularly useful in
  command files.  To continue any command to another line, end the
  first line with an ampersand (&).  Commands can be continued in
  this way to any number of lines, limited only by the total command
  length limit of 4096 characters.  The end-of-line counts as a blank
  for command parsing purposes.  A comment is indicated by beginning
  the command with two dashes (--).  A completely blank command is
  also a legitimate comment.  Comments are not allowed between lines
  of a continued command.

EXAMPLES
  do latr.fit
  do /user/maine/someFile

ERROR HANDLING
  If the specified command file can not be read (usually because you
  gave the wrong name), an error number is printed and control
  returns to the user.  The "sys files" command can be useful in this
  situation to verify the file name.
```

SEE ALSO
  sys

IMPLEMENTATION
  Internal command within the getCmd subroutine.

KEYWORDS
  do/execute a command file,
  command line form/syntax, continuation lines, comment lines

AUTHOR Richard Maine - NASA Dryden
VERSION 1.1
DATE 1/23/85

A.2.3 *Help* Command—

```
help [cmd] -- help command

USAGE
  help [<command_name>]

DESCRIPTION

  Gets help on commands in this program.  This version of help
  is set up to look only for help on this program.  It also
  accepts all arguments described in the system helpfile (do
  "sys help help").

  |------------------------------------------------------------|
  |        TO GET A LIST OF THE AVAILABLE COMMANDS,            |
  |                 DO "HELP COMMANDS".                        |
  | Some programs also have helpFiles for variables or topics, |
  | which you can find with "help variables" or "help topics". |
  |------------------------------------------------------------|

EXAMPLES
  help quit

SEE ALSO
  sys, sys help help

IMPLEMENTATION
  Calls the system help utility, with the search rule set for
  this program.

KEYWORDS
  help command

AUTHOR Richard Maine - NASA Dryden
VERSION 1.1
DATE 1/22/85
```

A.2.4 *Method* Command—

```
method [cmd] -- define interpolation methods

USAGE
  method  inSig1=meth1 inSig2=meth2 ...

PARAMETERS
  inSig?
    Name of an input or filtered parameter.

  meth?
    Interpolation method for the specified input signal.  Allowable
    interpolation methods are hold (meaning hold-last-value) and
    interpolate (meaning linear interpolation).  Anything beginning
    with h or i will be accepted as an abbreviation.

DESCRIPTION
  This command specifies the methods to be used for interpolating
  signals to common output times.  This command overrides, on a
  signal-by-signal basis, the default interpolation method for each
  file specified in the read command.  Any signal not mentioned in a
  method command uses the default interpolation method specified in
  the read command; if the read command did not specify the method
  either, hold-last-value interpolation is used.

  The interpolation method is applicable only to input or filtered
  signals.  Calculated functions and output parameters are always
  evaluated with the interpolated data at the output times.

  All method data is discarded whenever a read command is executed.
  Any applicable methods must be re-entered, even if they are the
  same as those in effect for the previous files.

  Anything begining with meth will be accepted as a synonym for the
  method command.

EXAMPLES
  meth alpha=h beta=interp &
       p=hold-last-value
```

CAVEATS

Hold-last-value interpolation is far more efficient than linear interpolation. For input files that have no active signals with skews or linear interpolation, the program uses a special-case fast algorithm. As soon as an input file has a single active signal skew or linear interpolation, the special algorithm no longer applies for that file and the performance becomes substantially worse.

Linear interpolation is meaningless for parameters such as digital words. The program has no idea which parameters are in this category; it will obediantly trash such parameters if you ask it to.

ERROR HANDLING

Methods not beginning with h or i will cause an error message, leaving the previous method specification (if any) unchanged.

SEE ALSO

read, skew, show, copy

KEYWORDS

method/meth command,
specify/set/select/define/change interpolation/synchronization/sync methods

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 29 Jun 87

A.2.5 *Quit* Command—

```
quit [cmd] -- exit the program normally

USAGE
  quit

DESCRIPTION
  Terminates the program and returns to the operating system.

SEE ALSO
  sys

KEYWORDS
  quit command,
  quit/exit/terminate the program

AUTHOR Richard Maine - NASA Dryden
VERSION 1.1
DATE 1/23/85
```

A.2.6 *Read* Command—

```
read [cmd] -- specify input data file(s)

USAGE
  read fileName
       fSkew=<file_skew>
       +hold

PARAMETERS
  fileName
    Name of the input file.  This is a required parameter.

  fSkew
    Time skew to be added to all times on this file, in seconds.
    This skew is independent of the individual signal skews.  If
    signal skews are specified, they are in addition to the file
    skew.  If all signals in a file are to be skewed by the same
    amount, it is FAR more efficient to specify this as a file skew
    than to specify all the individual signal skews with the skew
    command.  The file skew may be arbitrarily large and has no
    impact on efficiency.  The default is 0.

  +hold(-interpolate)
    Default interpolation method for signals in this file.
    Hold-last-value interpolation is specified by +hold; linear
    interpolation is specified by +interpolate.  This default can be
    overridden on a signal-by-signal basis using the method command.
    If unspecified, the default is +hold.

DESCRIPTION
  This command specifies the data files to be read.  It also
  specifies some details about how the files will be processed.

  The read command does not actually read any data from the files; it
  just opens the files and prepares them for reading.  The actual
  data must subsequently be read using the copy command.

  Each execution of the read command first closes all previously open
  input files.  To merge data from multiple input files, you must
  specify them as a list of files in a single read command.  The list
  syntax requires you to specify the name of the each file, followed
  by all parameters relevant to the that file.  A comma indicates the
  end of the specifications relating to each file.
```

Any previously-defined output signal definitions remain unchanged when a read command is executed. The program will relink the output signals to the available inputs on the new input files. Any previously open output data file remains open. This allows convenient splicing of time segments from multiple input files onto a single output file. A typical command sequence to do such splicing would be:

```
-- copy relevant times from the first input file.
read file1
signals +all
write outFile
copy time 1:0:0:0-2:0:0:0
-- copy data from second input file to same output file.
read file2
copy time 2:0:0:0-3:0:0:0
```

All previous information about skews and interpolation methods is discarded when a read command is executed. These data must therefore be respecified even if they are the same as for the previous file(s).

EXAMPLES

```
read datafile
read file1 fSkew=.05, file2 fSkew=-.02, file3
```

ERROR HANDLING

Any errors in parsing the command or opening the files will cause all the input files to be closed.

If any signals needed for computing the currently-defined output signals are missing, an error message will be printed and the corresponding output signals will be set to 0.

SEE ALSO

signals, write, copy, show, sys files

KEYWORDS

read command,
specify/set/select input data file
names/skews/(syncronization/interpolation methods)

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 3 Sept 86

A.2.7 *Show* Command—

```
show/list [cmd] -- list signal names

USAGE
  show
or
  list

DESCRIPTION
  Shows the currently-defined input, calculated and output signal
  names.  Calculated signals that require unavailable inputs are
  shown in parens.  Show and list are synonyms.

EXAMPLES
  show

CAVEATS
  The list tends to be long and scroll off the screen.  Use ^S/^Q to
  pause and restart it.  There is no way to abort the list short of
  aborting the program.  Probably ought to provide parameters to ask
  for specific portions of the data.  This command will probably be
  expended in the relatively near future, possibly before production
  release.

SEE ALSO
  signals

KEYWORDS
  show/list command,
  show/list selected input and output signal names

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 27 Aug 86
```

A.2.8 *Signals* Command—

```
signals [cmd] -- define signals to be written

USAGE
  signals [+all|+add|+delete] outSig1[=expr1] outSig2[=expr2] ...

PARAMETERS
  At most one of the switches +all, +add, or +delete may be specified
  in a single signals command.  Furthermore, this specification must
  be the first argument of the command or it will not be recognized.
  If none of these switches is specified, the specified signals
  completely replace any previous list.

  +all
    If +all is specified, the output signal list will be set to
    consist of all currently available signals.  There is no
    computation or renaming of signals.  Any subsequent arguments
    will be ignored if +all is specified.

  +add
    If +add is specified, the specified signals are added to the list
    established by previous signals commands.

  +delete
    If +del is specified, the specified signals are deleted from the
    list established by previous signals commands.  Anything begining
    with +del is accepted as an abbreviation.  If +delete is
    specified, the expressions in the signal list are irrelevant.

  outSig?
    Each outSig parameter defines the name of a signal to be written.
    Signal names must not contain commas, equal signs, quotes (single
    or double), parentheses, or embedded or leading blanks.  It will
    probably simplify your life if you also avoid the characters '+'
    and '-' and if you start each variable name with a letter, but
    these suggestions are not enforced.  Signal names are limited to
    16 characters.  The outSig names can not be quoted.

  expr?
    The expr parameters are expressions defining the computation of
    the output signals.  Expressions can consist of up to 5 terms in
    the forms:
      <sign><constant>
      <sign><signal-name>
      <sign><constant>*<signal-name>
    where
      <sign> is + or - (may be omitted from first term)
```

    <signal-name> is the name of an input or calculated signal.
      (This includes only calculations defined by calculated
      function subroutines, not calculations defined by the signals
      command). Signal names follow the same rules as for output
      signals. Signal names may be enclosed in quotes (either
      single or double quotes, but they must match). If a signal
      name.contains '+' or '-' characters, or if it starts with a
      digit or dot, then it must be enclosed in quotes.

   If the expression is omitted, the output signal is assumed to be
   equal to an input or calculated signal of the same name (enclosed
   in quotes).

DESCRIPTION

This command defines the names of the signals to be written and how they are to be computed. The computations allowed include selection or renaming of an input signal, plus simple linear combinations of input signals. In this context, input signals include those obtained from filters or calculated functions.

Anything begining with sig will be accepted as an abbreviation for the signals command.

At least one signals command must precede the first write command. You should not normally use the signals command while an output file is open. The signal names on an output file are determined by the signal selection effective at the time the write command for that file is issued. Any subsequent signals commands will splice different signals onto the same channels of the output file; this has subtle implications and is appropriate only for special applications. A warning message will be issued if you attempt this.

EXAMPLES

```
signals +all
sigs +add p q r
signals +del alpha q-bar
signals &
  alpha-c="alpha-f"-3.125*q  tap3="003" &
  q-bar=x12345  deAvg=.5*"de-left"+.5*"de-right"
```

CAVEATS

The expression parser is quite crude; do not be confused by its similarity to Fortran syntax. It can not handle any forms other than those listed. For instance, the multiplying constant must always precede the signal name instead of following it. Exponent form (e.g. 1.e-3) is not accepted for constants. Parentheses are not recognized. There can be no blanks in an expression, except around the equals sign.

The parser does not strictly enforce the rules for signal name syntax in all contexts. You can sometimes get by with expressions not meeting the stated syntax rules. For instance 1*3 is interpreted as a constant 1 times a signal named 3, even though the 3 is not quoted. Such expressions are confusing and are not guaranteed to work with future parsers. I advise avoiding them.

The only optimization of the expressions is a special case for expressions consisting of a single signal name with the multiplying constant omitted or equal to +1.0. (The large majority of expressions have this form). Expressions such as 2+2 will work, but the addition will be repeated every time point, which is a horrible waste of computer time.

BUGS

The signal name 1.0 is reserved for internal use. If you have an input signal with this name, references to that input signal will not give the correct results.

The program does not currently detect the occurance of multiple output signals of the same name, but files with such duplicate names may cause difficulties in your future life. If there are multiple input signals with the same name, there is no way to specify which one you want; the result is not guaranteed to be repeatable.

ERROR HANDLING

Ill-formed expressions will be give an error message and substitute a blank expression, which will give the value 0.

If any of the signals used in the computation of an expression is unavailable, an error message will be printed and the value 0 will be used for that signal. Although currently inactive, the expression will be remembered and may become valid after a subsequent read command.

```
SEE ALSO
  read, write, show, copy

KEYWORDS
  signals/sigs command,
  specify/set/select/define/change/rename signal/channel names
  and computations/calculations

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 20 Nov 86
```

A.2.9 *Skew* Command—

```
skew [cmd] -- define input signal skews

USAGE
  skew  inSig1=skew1 inSig2=skew2 ...

PARAMETERS
  inSig?
    Name of an input or filtered parameter.

  skew?
    Time skew for the specified input signal, in seconds.  This
    skew is added to the time tag of the measurement.  Thus a
    positive skew value adds lag to the signal (possibly to
    compensate for a lead in the raw data).  This skew is in
    addition to any file skew specified in the read command.

DESCRIPTION
  This command specifies the signal skews to be added to the time
  tags of the input signals.

  Note that skews can not be applied to calculated functions or
  output parameters.  Calculated functions and output parameters are
  always evaluated with the skewed input data; thus calculated
  functions can be indirectly skewed by skewing all of their input
  signals.

  The output data is always written in frames of data interpolated to
  common times.  This interpolation is done either by linear
  interpolation or hold-last-value, as specified by the method
  command.  Note that a skew smaller than the sample interval can
  sometimes have no net effect on the output for signals using
  hold-last-value interpolation.

  All skew data is discarded whenever a read command is executed.
  Any applicable skews must be re-entered, even if they are the same
  as those in effect for the previous files.

  Anything begining with skew will be accepted as a synonym for the
  skew command.

EXAMPLES
  skew alpha=.04  beta=-.03 &
       p=.01
```

CAVEATS

There are significant performance penalties for processing skews, and these penalties become larger as the skew becomes larger. For input files that have no active signals with skews or linear interpolation, the program uses a special-case fast algorithm. As soon as an input file has a single active signal skew or linear interpolation, the special algorithm no longer applies for that file and the performance becomes substantially worse.

There are also limits to the magnitudes of signal skew that can be applied. These limits are functions of several factors and can be increased if needed (but this will cause further performance degradation). There will be a warning message if you exceed the limits.

These limits do not apply to the file skews specified in the read command. The file skews can be arbitrarily large and have no performance implications. Thus, if the same skew is to be applied to all signals in a file, it is far more efficient to specify it as a file skew than as individual signal skews.

It is easy to get the sign of the skew wrong. If you want to skew the data to correct for a lag in the sensor, you must specify a negative skew.

ERROR HANDLING

Ill-formatted skew values will cause an error message, leaving the previous skew (0 if never specified) unchanged.

SEE ALSO

read, method, show, copy

KEYWORDS

skew command,
specify/set/select/define/change signal/channel time skews

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 29 Jun 87

A.2.10 *Sys* Command—

```
sys [cmd] -- execute a system command without exiting program

USAGE
  sys <system_command_line>

DESCRIPTION
  Sys allows the execution of any system command line from within the
  program.  The command line need not be quoted.  Common uses include
  the system files and to commands.  The synonym system (or anything
  else beginning with sys) is accepted.

EXAMPLES
  sys files
  sys to monty "I am busy now"

ERROR HANDLING
  If the command fails, any error messages will be printed and
  control will be returned to the program.

IMPLEMENTATION
  Calls $Shell, with appropriate error handling.

KEYWORDS
  run a system command, sys command

AUTHOR Richard Maine - NASA Dryden
VERSION 1.1
DATE 1/23/85
```

A.2.11 *Write* Command—

```
write [cmd] -- specify output data file name

USAGE
  write [filename] [format]

PARAMETERS
  fileName
    Name of the file to be written.  If no name is supplied, the
    previous output file will be closed without yet opening a new
    one.

  format
    File format to be used.  Currently accepted values are unc2,
    cmp2, ascl, and lisl.  If omitted it defaults to cmp2.

DESCRIPTION
  This command specifies the data file to be written.  It closes any
  previously open output data file and opens a new file with the
  specified name and format.

  The write command does not actually write any data to the output
  file; it just opens the file and prepares it for writing.  The
  actual data must subsequently be written using the copy command.

  The signals to be written must have been specified before executing
  the write command.  Any subsequent execution of a signals command
  will splice different signals onto the same channels of the output
  file; this is appropriate only for special applications.

  The interpretation of the format parameter depends on the write
  routines.  It is possible for the write routines to ignore this
  parameter or change its interpretation.  With the default write
  routines, the possible values are:
    unc2: uncompressed 2 format.
    cmp2: compressed 2 format.
    ascl: ascii 1 format. (primarily for tape transfer to other
          machines).
    lisl: listing format. (for screen or printer listings only;
          no read routines for this format are supported).
```

Note that you can list directly to the terminal screen by specifying $stdout (Elxsi-specific) as the filename. The resulting display, however, will not stop at the end of each screenful; you must use ^S/^Q to start and stop the display if desired. Only asc1 and lis1 formats will work to the screen (the others are binary formats); the lis1 format is more readable.

EXAMPLES

```
write dataFile
write dataFile unc2
write $stdout lis1
```

CAVEATS

the signals command must have been executed prior to the write command in order to specify the signal names to be written.

ERROR HANDLING

If no output signals are defined, or if any errors in parsing or execution occur, the output file will be closed.

SEE ALSO

read, signals, copy, show, sys files

KEYWORDS

write command,
specify/set/select/close output data file name

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 3 Sept 86

A.3 Topic Help Files

A.3.1 Calculations—

calculations [topic] -- calculated functions in getData

DESCRIPTION

This helpFile gives an overview of calculated functions in getData.

There are 3 different means of defining calculated functions in getData: the signals command, the calculated function subroutines, and the filter subroutines.

The signals command allows you to define some simple calculations interactively, without writing Fortran code. You can interactively define calculations that are simple linear combinations of input signals. This includes such common functions as averages, differences, sign changes, plus general bias and scale factor corrections.

These calculations can use input file signals, filtered signals, or calculated function signals. A calculation defined in the signals command can not use another calculation defined in the signals command. These calculations are performed immediately before writing each output record; they have no intrinsic sample rates. The input signals used in the calculations are all skewed and interpolated to the output times before the calculations are performed. For details, see the helpFile for the signals command.

Calculations more complicated than supported in the signals command must be implemented by Fortran subroutines. Up to 5 independent calculated function modules can be simultaneously installed (this number can be easily increased if needed). The calculated function modules are called CF1 to CF5. Each calculated function module normally implements multiple calculated functions.

Each calculated function module can use signals from the input files, the filter module, and lower-numbered calculated function modules. It can not use signals from higher-numbered calculated function modules or signals defined in the signals command. The calculations are performed immediately before writing each output record; they have no intrinsic sample rates. The input signals used in the calculations are all skewed and interpolated to the output times before the calculations are performed.

Each calculated function module is defined by a set of 3 Fortran subroutines, called allocateCFx, activateCFx and doCFx, where the x is replaced by the calculated function module number (1-5). For details, see the helpFiles for these routines. (The helpfile names do not include the x suffixes).

Filters can not be conveniently implemented in the normal calculated function modules because digital filters are inherently linked to specific sample rates, whereas the normal calculated function modules do not have inherent sample rates and may be called at different rates, depending on the requested output. Therefore, separate provision is made for filtered signal computations. Only a single set of filter routines is currently allowed; this set of routines can support multiple filters.

The filters can use only signals directly from an input file. Other calculated functions can not be filtered (though they can use filtered inputs, which normally achieves about the same effect). The input signals used for the filter are raw, without skew corrections or interpolation. The filtered result may be skewed and interpolated in the same way as signals read from the input files. (Normally, you would expect to use a different skew for the filtered signal than for the raw signal anyway).

The filter interface conveniently allows only recursive causal filter forms; i.e., the filters can depend only on prior and current data, not on future data. There is no easy way to run forward/backward filters or smoothers. Note that you can skew the filtered result to approximately compensate for the group delay of the filter.

The filters are defined by the subroutines allocateFilt, activateFilt, reMapFilt, and doFilt. For details, see the helpFiles for these routines.

USAGE

The interactively-defined calculated functions are defined and accessed through the signals command.

The calculated functions defined by calculated function modules or the filter module are accessed interactively by signal name in the same manner as the input signals. The only difference is that you can not specify skews or interpolation methods for signals defined by calculated function subroutines; the calculations are done after all skews are applied. You can specify skews and interpolation methods for filtered parameters.

EXAMPLES

See the signals helpFile for examples of interactively defined calculations.

Source for a sample set of calculated function subroutines is in the files sample.CF1.f and sample.CF1.com in the /user/maine/getData.3.1/source directory.

Source for a sample set of filter subroutines is in the files sample.filt.f and sample.filt.com in the /user/maine/getData.3.1/source directory.

CAVEATS

The interface to the filter module is not as "clean" as I would like it to be. Unfortunately, performance requirements forced some compromises here. If I get some better ideas, this interface might change further in the future. The interface to the calculated function modules is much more "solid." Luckily, there seem to be only a few users of the filter module.

SEE ALSO

signals [cmd]
allocateCF,activateCF,doCf [sub]
allocateFilt,activateFilt,reMapFilt,doFilt [sub]

KEYWORDS

calculations topic,
(calculated function)/filter subroutines

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 8 Sept 86

A.3.2 CPUTime—

```
cpuTime [topic] -- cpu time estimates for getData on Elxsi

DESCRIPTION
  This helpFile gives cpu time estimates for getData.3.1.1
  running on an Elxsi 6400.

  Unless otherwise specified, data are based on tests with
  optimized cmp2 format read/write routines and with optimized
  doCopy routines in getData.  There is relatively little
  difference between cmp2 and unc2 format times in most cases.
  All times are quoted per frame (time point).  Times do not
  include setup overhead that is not repeated each frame.

  ----------------------- Overhead costs ---------------------
  Overhead Costs.
    .2 ms, with major parts  as follows:
      syncTo:  .05 ms
      doCalcs: .03 ms
      mapOut:  .02 ms
      nextT:   .03 ms

  ------------------ Input file positioning costs ------------
  For cmp2 format input files, there is a significant cost for
  positioning input files to the begining of the interval to be
  processed.  This cost is
    .25 ms per time point skipped, plus
    2 us per compressed data value skipped.
  You can estimate the number of compressed data values in a
  file as about one third of the file size in bytes.  Then
  multiply this by the fraction of the file you are skipping
  over.

  The file positioning cost for unc2 format files is negligable.
  I have not measured it for ascl format files, but it will be
  very large.

  ---------------------------- Input costs --------------------
  For each input file, there is an initial cost of
    .6 ms
  For each signal on an input file, whether used or not, add
    9 us
    This figure may vary from around 5 to 15 us with cmp2 files,
    depending on the compression.  The fastest reading is from
    highly compressed files.  The figure quoted is a typical
    average.
```

```
If the input file is asc1 format, add
  .15 ms per signal + .4 ms per line
  (3 signals on the first line, 4 on subsequent lines.)
For each signal used, independent of the format, add an
additional
  6 us

-------------------------- Output costs ----------------------
There is an initial cost of
  .6 ms
For each signal written, add
  8 us
  This figure may vary from around 5 to 15 us with cmp2 files,
  depending on the compression.  The fastest writing is to
  highly compressed files.  The figure quoted is a typical
  average.
If the output file is asc1 format, add
  .15 ms per signal + .5 ms per line
  (3 signals on the first line, 4 on subsequent lines.)
If the output file is lis1 format, add
  .25 ms per signal + .4 ms per line
  (5 signals on each line)

------------------------ Processing costs --------------------
For each signal used, also add
  6 us, divided as follows
    syncTo:  4 us
    mapOut:  2 us
    Depending on precisely how the signal is used, some of
    these components may not apply.

If ANY used signal on an input file has a non-zero skew or
uses meth=interp, then add
  25 us for every used signal on that file.

For each used signal with meth=interp, add
  5 us

For each skewed signal, add
  11 us per time point or fraction of a time point of skew

For each output calculation term specified in the signals
command, add
```

```
  10 us
  This need not be added for the first term of each signal,
  provided that term has no multiplier.  Examples:
    outsig=insig  (add nothing)
    outsig=2*insig (add 10 us)
    outsig=insig+3 (add 10 us)
    outsig=.5*insig1+.5*insig2 (add 20 us)

I have not done time testing for calculated function routines.

-------------------------- Filter costs ----------------------
Filter costs, of course, depend heavily on the specific filter
implementation.  The following estimates are for a typical
filter consisting of a 3rd order lowpass plus a notch.  The
figures are based on tests with the optimizer used on the
filter routines.

For having the filter routines installed in the program,
whether used or not, add
  40 us for each input file that has filters defined

For each filter used, add
  50 us

A filter does not count as a signal read from an input file;
however, the unfiltered signal must be read from the input
file in order to be filtered.  The input times for the needed
unfiltered signals must therefore be included.  The program
implementation also forces both the filtered and unfiltered
signal to be processed by syncTo, even though the unfiltered
signal might not really need this processing.

An indirect cost of filtering is that filtered signals usually
compress very poorly.  Poor compression can increase the time
required for output (in addition to the rather obvious
increase in the file size).
```

EXAMPLES

```
  Read 4 signals from a cmp2 format file having 685 signals.
  Write in cmp2 format.  No skewing or interpolation.  Copy all
  16015 times from the file.

  overhead                      .2 ms
  input .6 ms + 685*9 us = 6.8 ms
  output .6 ms + 4*8 us  =  .6 ms
  processing 4*6 us      =  .0 ms
  total                    7.6 ms * 16015 frames = 122 sec
```

Note that the large majority of the time of this example is
from the large number of unused signals in the input file.

CAVEATS

Times will vary somewhat as a function of the data, system
load and other factors not considered in the tests. The
estimates can not be trusted to better than about 10-20%; some
cases may vary more.

Any extrapolation of these estimates to other machines is at
your own risk.

KEYWORDS

cpuTime topic,
estimating cpu time for getData

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 19 Sept 86

A.3.3 FileInterFace—

```
fileInterface [topic] -- time history file interface routines

DESCRIPTION
  This helpfile describes the time history file interface
  modules.  The time history file interface modules are meant to
  provide compatable access to a variety of data file formats.
  These modules are particularly oriented around flight time
  history data files.  A program using these interface modules
  can be modified to read or write different file formats by
  merely rebinding with different interface modules.  No program
  changes are required.  The use of these interface modules thus
  avoids the necessity to modify each program to access special
  file types.  A single specialized interface module can serve
  for all programs designed to use these modules.

  You can often get by without even rebinding, because there is
  a file read module that automatically recognizes and reads
  several formats, plus a file write module that will write in
  any of these same formats.

  Currently supported formats include:
  unc1 - uncompressed 1 format.  A simple format similar to
         "mmle" format as used on the CDC.  Used mostly for
         compatability and for areas where simplicity of format
         is an overriding factor.  Support for this format is
         limited and available only by binding in a special set
         of read and write routines.
  unc2 - uncompressed 2 format.  An improvement on unc1 format,
         which adds header information including signal names.
         Fully supported.
  cmp2 - compressed 2 format.  A format using data compression.
         The usual choice for large files.  Fully supported.
         Elxsi-specific.
  asc1 - ascii 1 format.  Used mostly for tape transfer to other
         systems.  This format is inefficient both in file size
         and access time.  It is not recommended for internal
         Elxsi use.  Fully supported.
  lis1 - listing 1 format.  Used only for screen and printer
         listings.  Fully supported for writing.  No read
         routines are supported for this format.  The exact
         details of the format are subject to change.
```

USAGE

The sequence of subroutine calls to write a file using the time history file interface routines is as follows:

```
openW - to open the file for writing.
fWrite...fWrite - called repeatedly to write records.
closeW - to close the file.
```

The sequence of calls to read a file using the interface routines is:

```
openR - to open the file for reading.
rSigs (optional) - to find names of the available signals.
sigsR or chansR (optional) - to specify the signal names or
                             channel numbers to be read.
rewR (optional) - to position the file at the first record.
fSeek (optional) - to position the file after a requested
                   time and read a record.
fRead - to read the next record on the file.

closeR - to close the file.  (Can usually be safely omitted,
         but is recommended).
```

The subroutines between openR and closeR can all be called any number of times and in any order, except for rSigs. Some versions may not perform as expected if rSigs is called after any calls to fRead or fSeek.

LIBRARY

Jobs using a particular set of file interface routines must bind in the object file containing the appropriate routines. In addition, most of the interface libraries use subroutines in /user/maine/lib/misc.lib.o.

The interface subroutines are in the directories /user/maine/fRead and /user/maine/fWrite. The supported fWrite object file is /user/maine/fWrite/auto.o. The supported fRead object file is /user/maine/fRead/auto.o. Source, but not object, code for some simpler, more portable versions is also maintained.

CAVEATS

Most of the routines have dimension limits on the number of allowable channels on a file. Typical current dimensions allow up to 1000 channels per file. This can be easily changed if required.

The routines are not set up conveniently to be called from programs compiled with the +double switch. All floatting point quantities are 64-bit precision, but integers and logicals are only 32 bits. If you call these routines from programs compiled with +double, you must explicitly declare any integer or logical quantities used as arguments to be integer*4 or logical*4.

SEE ALSO

particular subroutines and file formats

SUBROUTINES

A full set of read access routines includes:
openR,closeR,rSigs,sigsR,chansR,rewR,fRead, and fSeek.
A full set of write access routines includes:
openW,closeW,fWrite

KEYWORDS

time history data file read/write/access/interface
subroutines/routines,
openW,closeW,fWrite,
openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek

AUTHOR Richard Maine - NASA Dryden
VERSION 2.1
DATE 12/27/85

A.3.4 Version—

```
version [topic] -- version 3.1 changes to getData

DESCRIPTION
  GetData version 3.1 is now released.  This is a major rewrite
  of getData to add new capabilities, with large portions of the
  program rewritten from scratch.

  Everything relating to this version currently lives in the
  directory /user/maine/getData.3.1.  To access version 3.1, use
  the command "/user/maine/commands/getData".  To access the
  bind shellfile for version 3.1, use the command
  "/user/maine/commands/bindGetData".

USAGE CHANGES/INCOMPATABILITIES
  As long as you are not using the new capabilities, the general
  usage of the program is quite similar to that of the previous
  version.  There are some changes in detail, outlined in the
  following.  See the internal helpFiles for precise details.

  The change that will most immediately affect everyone is in
  the default for the signals command.  In the prior version,
  the output signals defaulted to all the signals available from
  the first read command, which was often convenient but
  occasionally awkward for read commands after the first.  In
  version 3.1, there are no output signals selected by default.
  This means that a signals command is now mandatory.  The
  command 'signals +all' will duplicate the effects of the
  default in the previous version.

  The ability to specify expressions for output signals means
  that some signal names now cause ambiguity in some contexts.
  In particular, signal names begining with a digit and signal
  names containing "-" characters can cause problems; there are
  numerous existing files with such signal names.  You must
  quote any such ambiguous signal names appearing on the right
  side of the equals sign in the signals statement.  If you do
  not have any equals signs in the signals statement (i.e.  if
  you do not use it to rename signals), this change does not
  affect you.  See the signals helpFile for details.

  Multiple files specified on a single read command must now be
  separated by commas.  Previously, blanks or commas were
  acceptable.  This change is to allow for some extra optional
  syntax in the read command.
```

The dt parameter on the copy command now causes drastically different behavior than before. If you specify dt, the output file will now have exactly the specified dt, interpolating the input data as needed. In the previous version, you could use either the thin parameter or the dt parameter to specify thinning (though there were some subtleties in the use of dt). In the current version, you get quite different results from specifying dt than from specifying thin. The thin parameter still specifies simple thinning.

NEW CAPABILITIES

The most important new capabilities are time-skewing and interpolating data. With these capabilities, getData can now do essentially everything that the sync program could do. Either linear interpolation or hold-last value interpolation can be selected on a signal-by-signal basis. See the discussions in the helpFiles on the new "skew" and "method" commands, plus the added options in the "read" command. Be warned, however, that invoking this capability causes large increases in the required computer time and memory.

Closely related to the interpolation capability is the new capability to force the output file to have constant specified sample intervals. The data are interpolated to the required output times. This capability is important for some analysis programs that can not handle data dropouts or other timing irregularities. See the helpFile for the copy command, particularly the dt parameter.

Another major new feature is the ability to define some simple calculations interactively, without writing Fortran code. You can interactively define calculations that are simple linear combinations of input signals. This includes such common functions as averages, differences, sign changes, plus general bias and scale factor corrections. See the helpFile for the signals command. More complicated calculations still require Fortran coding.

There are 2 new switches in the signals command: +all and +delete. These allow such things as the much-requested capability to change a single signal name without also entering the entire list of unchanged signals.

The program is significantly faster (except when the skew and interpolation options are used) in some cases. There is an overall speedup from the use of the optimizer. (See the caveats below). This typically seems to gain about 20-30%. There have also been some algorithm changes that significantly speed up cases where a small number of parameters are being read from a file with many parameters. The speedup is sometimes as large as a factor of 2 in extreme cases. I should note that it is still far slower than reading the same parameters from a smaller file. Files with more than one or two hundred parameters are inefficient and likely to remain so, but the new algorithm reduces some of the efficiency penalty. (You might consider splitting up such files).

SUBROUTINE INTERFACES

The interface to the calculated function subroutines has changed for two reasons. First, the old interface caused unacceptable performance penalties for the skewing and interpolation options. The old interface provided no way for the program to know what signals were actually needed, so all available signals had to be skew-corrected and interpolated.

Second, as long as I was changing the interface anyway, I added the capability to have multiple independent calculated function modules installed at the same time. Thus, for example, a user could add his own calculated functions in addition to those defined and supported by the project. Formerly, the easiest way to do this was to make 2 separate getData runs, creating an intermediate file. (Sorry, this capability does not apply to filters; only one filter module can be installed in a single job).

The biggest effect is on the setupCalcs routines. Few changes will need to be made in most versions of doCalcs. See the "calculations" topic helpFile for details. There are sample calculated function routines in the files sample.CF1.f, sample.CF1.com, sample.filt.f and sample.filt.com in the directory /user/maine/getData.3.1/source.

DOCUMENTATION
This file is available within getData by typing "help version". All internal helpFiles have been substantially revised. HelpFiles have been added for the calculated function subroutines. A written manual for this version will be prepared "soon."

CAVEATS
Large skews can eat up prodigious amounts of computer time with this version; please exercise appropriate constraint.

SEE ALSO
Internal helpFiles.

KEYWORDS
version topic,
getData version 3.1/3.1.1 changes

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 5 Sept 86

A.4 Calculated Function Subroutine Help Files

A.4.1 Subroutine *AllocateCFx*—

```
allocateCF [sub] -- locate signals for calculated functions

USAGE
  call allocateCFx

DESCRIPTION
  This subroutine defines and locates the signals used in the
  calculated function routines.  It also defines and allocates
  the signals to be calculated.  There is one allocateCFx
  routine for each calculated function set, with x replaced by
  the calculation set number (1 to 5).  It is called before any
  calls to activateCFx or doCFx.  It may be called multiple
  times in a single run if multiple read commands are executed.

  An allocateCFx routine should have 4 sections, performing the
  operations described below.  Much of the actual work is done
  in subroutines and functions called by allocateCFx.  All of
  the subroutines and functions mentioned are provided
  independently of the user-written calculated function
  subroutines.

  AllocateCFx will need one or more common blocks to pass data
  to subroutines activateCFx and doCFx.  I suggest the common
  block name /CFx/ for this purpose (with x replaced by the
  calculation set number).

  1. Declare a descriptive label (up to 60 characters) for the
     calculation set by calling subroutine labelCalc.  LabelCalc
     has 2 arguments: the calculation set number and the label.
     Example:

       call labelCalc(1,'CF1 sample. Richard Maine. 12 Aug 86')

  2. Find all input signals needed for the calculations by
     calling function sigChan.  SigChan has one argument: the
     signal name.  It returns an integer channel number for the
     signal.  A channel number of 0 means that the signal was
     not found.  You should save these channel numbers in common
     block /CFx/, as they will be needed by subroutines
     activateCFx and doCFx.  I begin the variable names for
     these numbers with an "i" to remind me that they are input
     signals, but no naming conventions are enforced.  Example:
```

```
    iDeR = sigChan('deR')
    iDeL = sigChan('deL')
    iQbar = sigChan('qBar')
```

3. Allocate channel numbers for all signals that are defined by this calculation set by calling function calcChan. CalcChan has one argument: the signal name. It returns an integer channel number for the signal. A channel number of 0 means that the channel could not be allocated for some reason (possibly dimension limits or a name conflict). You should also save these channel numbers in common block /CFx/ for use by subroutines activateCFx and doCFx. I begin the variable names for these numbers with an "o" to remind me that they are output signals, but no naming conventions are enforced. Example:

```
    oDe = calcChan('de')
    oDa = calcChan('da')
    oKeas = calcChan('keas')
```

4. Disable those calculations needing unavailable signals by calling subroutine cantCalc. Test for a channel number of 0 to determine if a signal is unavailable. CantCalc has a single argument: the channel number of the calculated signal. CantCalc will set this argument to 0 and will put parens around the signal name so that future calls to sigChan will not be able to find it. Defining and then disabling a calculation like this is preferrable to just bypassing the definition because the user will be able to see the signal name in parens, indicating that the calculation is installed but is missing some required inputs. Example:

```
    if (iDeL.eq.0 .or. iDeR.eq.0) then
      call cantCalc(oDe)
      call cantCalc(oDa)
    endif
    if (iQbar.eq.0) call cantCalc(oKeas)
```

It is permissable to make multiple calls to cantCalc for the same signal or for a signal that could not be allocated; such redundant calls will have no effect.

NOTES

If any calculated function is used as an input to another calculated function in the same CF routine, you must adhere to the following conditions to correctly maintain the interdependencies of the calculations. These conditions are automatically fulfilled for calculated functions used as inputs in higher-numbered CF routines.

To determine if a calculated result is available for use in another calculation, you test for a non-zero channel number for the former calculation. For this test to work correctly, it must be after any calls to cantCalc for the former calculation. For instance, the sequence:

```
  if (oDe.eq.0) call cantCalc(oAlphaTrim)
  if (iDeL.eq.0 .or. iDeR.eq.0) call cantCalc(oDe)
```

is incorrect because oDe is tested before a possible call to cantCalc for it.

EXAMPLES

The full text of the subroutines using the above examples is in the files sample.CF1.f and sample.CF1.com in directory /user/maine/getData.3.1/source.

CAVEATS

Dont forget to declare sigChan and calcChan to be integer; likewise for the output channel number variables if you follow my naming convention.

ERROR HANDLING

No special error treatment is needed other than that mentioned in the above description.

SEE ALSO

calculations [topic]
activateCF, doCF [sub]

KEYWORDS

allocateCF/allocateCFx subroutine,
allocate channel numbers for (calculated functions)/calculations

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 29 Sept 86

A.4.2 Subroutine *ActivateCFx*—

```
activateCF [sub] -- activate needed calculated functions

USAGE
  call activateCFx

DESCRIPTION
  This subroutine activates calculated functions and their
  inputs as needed for the following processing.  There is one
  activateCFx routine for each calculated function set, with x
  replaced by the calculation set number (1 to 5).  It is called
  after allocateCFx and before doCFx.  It may be called multiple
  times.  It will always be called at least once between any
  call to allocateCFx and subsequent calls to doCFx.

  An activateCFx routine should perform the 2 operations
  described below.  All of the subroutines and functions
  mentioned are provided independently of the user-written
  calculated function subroutines.

  ActivateCFx will need the channel numbers determined by
  subroutine allocateCFx and placed in common block /CFx/.  It
  will also need to pass activation flags to subroutine doCFx
  through this common block.

  1. Determine whether each calculated function is needed by
     calling function isUsed.  IsUsed has a single argument: the
     channel number of the calculated function.  It returns a
     logical true if the signal should be calculated; otherwise
     it returns false.  You should save these flags in common
     block /CFx/, as they will be needed by subroutine doCFx.  I
     begin the variable names for these flags with an "use", but
     no naming conventions are enforced.  Example:

       useDe = isUsed(oDe)
       useDa = isUsed(oDa)
       useKeas = isUsed(oKeas)

  2. For each needed calculation, declare that its input signals
     are also needed by calling subroutine setUsed.  SetUsed has
     a single argument: the input channel number.  Example:

       if (useDe .or. useDa) then
         call setUsed(iDeR)
         call setUsed(iDeL)
       endif
       if (useKeas) call setUsed(iQbar)
```

It is permissable and normal to call setUsed multiple times for the same signal. Although rarely useful, it is also allowed to call setUsed for a signal that is not available; such a signal will have been set to channel number 0, which always contains the data value 0.

NOTES

If any calculated function is used as an input to another calculated function in the same CF routine, you must adhere to the following conditions to correctly maintain the interdependencies of the calculations. These conditions are automatically fulfilled for calculated functions used as inputs in higher-numbered CF routines.

You should not call isUsed for any signal until after any possible calls to setUsed for that signal. The code for such situations should parallel the code in allocateCF, but in reverse order. For instance, if allocateCF has code like

```
if (iDeL.eq.0 .or. iDeR.eq.0) call cantCalc(oDe)
if (oDe.eq.0) then
  call cantCalc(oAlphaTrim)
  call cantCalc(oDeErr)
endif
```

then activateCF should have code like

```
uAlphaTrim = isUsed(oAlphaTrim)
uDeErr = isUsed(oDeErr)
if (uAlphaTrim .or. uDeErr) call setUsed(oDe)
uDe = isUsed(oDe)
if (uDe) then
  call setUsed(iDeL)
  call setUsed(iDeR)
endif
```

EXAMPLES

The full text of the subroutines using the above examples is in the files sample.CF1.f and sample.CF1.com in directory /user/maine/getData.3.1/source.

ERROR HANDLING

No errors should arise.

SEE ALSO

calculations [topic]
allocateCF, doCF [sub]

KEYWORDS
  activateCF/activateCFx subroutine,
  activate (calculated functions)/calculations

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 29 Sept 86

A.4.3 Subroutine *DoCFx*—

```
doCF [sub] -- evaluate calculated functions

USAGE
  call doCFx (time,data,reset)

PARAMETERS
  time: input, R*8
    Time of this frame, in seconds.

  data: i/o, R(*)*8
    Data vector for this frame.  Contains both input and output
    signals.

  reset: input, L*4
    Interval start flag.  This flag will be true on the first
    frame of each requested time interval; it will be false on
    all other frames.  This allows for the initialization of
    counters, integrators, etc.

DESCRIPTION
  This subroutine evaluates the calculated functions.  There is
  one doCFx routine for each calculated function set, with x
  replaced by the calculation set number (1 to 5).  It is called
  after allocateCF and activateCFx.  It is called one time for
  each output frame (record).

  DoCFx will need the channel numbers and activation flags
  placed in common block /CFx/ by subroutines allocateCFx and
  activateCFx.

  For each defined calculation, doCF should check the activation
  flag and perform the calculation if it is active.  The data
  vector has all the needed input signals, skew-corrected and
  interpolated to the output frame time.  The channel numbers
  are the indices into the data vector.  The calculation results
  are placed in this same vector, with indices given by their
  channel numbers.  Example:

    if (useDe) data(oDe) = .5*(data(iDeL)+data(iDeR))
    if (useDa) data(oDa) = .5*(data(iDeL)-data(iDeR))
    if (useKeas) data(oKeas) = 17.17*sqrt(max(data(iQbar),zero))
```

EXAMPLES

The full text of the subroutines using the above examples is in the files sample.CF1.f and sample.CF1.com in directory /user/maine/getData.3.1/source.

CAVEATS

Note that the data vector is dimensioned from 0, not from 1. The 0'th element has the value 0. Calculations that attempt to use unavailable signals will get this 0 value instead. In most cases, calculations that use unavailable signals will be disabled, but it is permissable to activate such a calculation, provided that 0 is an acceptable substitute input value.

Channels in the data vector that were not activated by calling setUsed are undefined. They are not guaranteed to have a 0 value or even a legitimate value at all. Don't use them; if you were going to, you should have called setUsed.

The values in the output signal channels of data are not guaranteed to be retained between calls. If you need to save an output value between calls, you must save it in a local or common variable.

Do not put any result in the data vector unless you have called isUsed for that signal and the result was true. Do not assume that just because you called setUsed, that isUsed must return true. (This assumption fails when setUsed is called for an unavailable signal, which is unusual but is legal). If you violate this rule, you might destroy the 0 value that is supposed to be stored in channel 0, causing havoc with other calculations. If two or more signals share much of the same computation, you may choose to compute all of the outputs whenever any of them are needed. However, do not place the results in the data vector without individually checking whether each result is used. For instance, don't write code like

```
if (uMach.or.uHp) then
  call airData(data(iPs),data(iPt),data(oMach),data(oHp))
endif
```

Instead, do something like

```
  if (uMach.or.uHp) then
    call airData(data(iPs),data(iPt),mach,hp)
    if (uMach) data(oMach) = mach
    if (uHp) data(oHp) = hp
  endif
```

ERROR HANDLING

There are no special provisions for error handling. The code should do whatever checks are necessary to assure valid execution, for instance to avoid taking the square roots of negative values. The code should not abort, except as a last resort. Error messages should also be avoided because they could become voluminous if repeated every time point. Reasonable error fixups include limiting values to valid ranges, setting special flag values or holding the previous value.

If you have to have an error message, consider logic to print it only on the first occurance in each time interval. The reset argument allows implementation of such logic. For example:

```
   if (reset) warned = .false.
   if (<condition>) then
     <fixup result>
     if (.not.warned) write (output,*) '*** oops'
     warned = .true.
   endif
```

SEE ALSO

calculations [topic]
allocateCF, activateCF [sub]

KEYWORDS

doCF/doCFx subroutine,
do/perform/evaluate (calculated functions)/calculations

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 29 Sept 86

A.5 Filter Subroutine Help Files

A.5.1 Subroutine *AllocateFilt*—

```
allocateFilt [sub] -- locate signals for filter

USAGE
  call allocateFilt (inF,inSig,nIn,maxOut,nOut)

PARAMETERS
  inF: input, I*4
    Input file number, from 1 to the maximum number of input
    files allowed.

  inSig: i/o, C(*)*16
    Vector of signal names for this input file.  On entry, it
    has the names of the signals available on the file.  On
    return, the names of the filtered signals for that file
    should be appended to the list.

  nIn: input, I*4
    Number of signals available on this file.  This is the
    number of valid names in inSig on entry.

  maxOut: input, I*4
    Maximum number of filters that dimension limits allow for
    this file.

  nOut: output, I*4
    Number of filters allocated for this file.  This should
    equal the number of names appended to the inSig vector.

DESCRIPTION
  This subroutine defines and locates the signals used in the
  filter routines.  It is called before any calls to
  activateFilt, reMapFilt or doFilt.  It may be called multiple
  times with the same input file number if there are multiple
  read commands.  Each call overrides any previous call for the
  same input file number.

  The allocateFilt subroutine should perform the following
  operations.

  1. Determine which filters go with this input file number.
     This is done by searching the inSig vector for the names of
     the unfiltered signals.  The function sIndex (provided) is
     used to do this search.
```

2. Append the names of the filtered signals for this file to the inSig vector. Set nOut to the number of filters allocated. Do not exceed the dimension limit given by maxOut.

3. Save the list of channel numbers for the unfiltered signals (obtained from function sIndex) and the filtered signals (allocated as you append to the inSig vector) in common /filtCom/. Subroutines activateFilt, reMapFilt and doFilt will need this data. Also, save a table that links the input file number to the appropriate allocated filters.

EXAMPLES

A sample set of filter subroutines is in the files sample.filt.f and sample.filt.com in directory /user/maine/getData.3.1/source.

CAVEATS

Dont forget to declare sIndex to be integer.

ERROR HANDLING

You should probably put out an error message if signals expected to be on the same file are not found together. You should certainly put out a message if filters are omitted because of the dimension limit. In either case, return normally after allocating those filters that you can.

SEE ALSO

calculations [topic]
activateFilt, reMapFilt, doFilt [sub]

KEYWORDS

allocateFilt subroutine,
allocate channel numbers for filters

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 8 Sept 86

A.5.2 Subroutine *ActivateFilt*—

```
activateFilt [sub] -- activate needed filters

USAGE
  call activateFilt (inF,iaOff)

PARAMETERS
  inF: input, I*4
    Input file number.

  iaOff: input, I*4
    Offset of this input file in concatenated data vector.

DESCRIPTION
  This subroutine activates filters and their inputs as needed
  for the following processing.  It is called after allocateFilt
  and before reMapFilt and doFilt.  It may be called multiple
  times.  It will always be called at least once between any
  call to allocateFilt and subsequent calls to reMapFilt or
  doFilt.

  An activateFilt routine should perform the 3 operations
  described below.  All of the subroutines and functions
  mentioned are provided independently of the user-written
  calculated function subroutines.

  ActivateFilt will need the channel numbers determined by
  subroutine allocateFilt and placed in common block /FiltCom/.

  1. Use the table defined by allocateFilt to match the input
     file number to the appropriate filters.

  2. Determine whether each filtered signal is needed by calling
     function isUsed.  IsUsed has a single argument: the channel
     number of the filtered signal within the concatenated
     vector.  To get this concatenated vector channel number,
     you must add iaOff to the filtered channel number allocated
     by allocateFilt.  IsUsed returns a logical true if the
     filtered signal is needed; otherwise it returns false.

  3. For each needed filter, declare that its input signal is
     also needed by calling subroutine setUsed.  SetUsed has a
     single argument: the input channel number within the
     concatenated vector.  As for the filtered signals, you must
     get this concatenated channel number by adding iaOff to the
     unfiltered channel number found by allocateFilt.
```

EXAMPLES
  A sample set of filter subroutines is in the files
  sample.filt.f and sample.filt.com in directory
  /user/maine/getData.3.1/source.

ERROR HANDLING
  No errors should arise.

SEE ALSO
  calculations [topic]
  allocateFilt, reMapFilt, doFilt [sub]

KEYWORDS
  activateFilt subroutine,
  activate filters

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 8 Sept 86

A.5.3 Subroutine *ReMapFilt*—

```
reMapFilt [sub] -- reMap filters to compressed locations

USAGE
  call reMapFilt (inF,iuMap)

PARAMETERS
  inF: input, I*4
    Input file number.

  iuMap: input, I*4
    Map from uncompressed locations to compressed locations.

DESCRIPTION
  This subroutine reMaps the channel numbers used by the filter
  subroutines.  It is called after each call to activateFilt,
  before any subsequent calls to doFilt.

  The channel numbers initially allocated in allocateFilt
  reserve channels for all signals available on each input file.
  For efficiency, the actual processing uses a data vector
  composed of only the signals needed, with the unused signals
  omitted.  The doFilt routine operates on this compressed data
  vector.

  Subroutine reMapFilt generates the channel numbers vectors
  used in doFilt by reMapping the original channel number
  vectors onto the compressed ones.  The compressed vectors must
  be distinct from the original ones instead of overwriting them
  because reMapFilt can be called multiple times after a single
  call to allocateFilt.  The iuMap vector gives the compressed
  channel number corresponding to each original channel number.
  Signals in the original vector that are omitted from the
  compressed vector are mapped to compressed channel 0.

  ReMapFilt will need the channel numbers determined by
  subroutine allocateFilt and placed in common block /FiltCom/.
  It will also need to pass the compressed channel numbers to
  subroutine doFilt through this common block.  The subroutine
  performs the following operations.

  1. Use the table defined by allocateFilt to match the input
     file number to the appropriate filters.  Then zero the list
     of used filters
```

2. Check each filter defined for that file to see if the channel number of its filtered signal maps to a compressed channel of 0. (Checking the return from isUsed ought to be equivalent, but you are about to use the mapped channel number anyway, so checking the channel number is more convenient and less prone to obscure errors). If the channel number maps to 0, skip that filter.

3. For each filtered channel that does not map to 0, increment the count of used filters for the file and save the mapped channel numbers for the filtered signal and its unfiltered source.

EXAMPLES

A sample set of filter subroutines is in the files sample.filt.f and sample.filt.com in directory /user/maine/getData.3.1/source.

ERROR HANDLING

No errors should arise.

SEE ALSO

calculations [topic]
allocateFilt, activateFilt, doFilt [sub]

KEYWORDS

reMapFilt subroutine,
reMap filter channel numbers

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 8 Sept 86

A.5.4 Subroutine *DoFilt*—

```
doFilt [sub] -- evaluate filters

USAGE
  call doFilt (inF,time,data,reset)

PARAMETERS
  inF: input, I*4
    Input file number.

  time: input, R*8
    Time of this frame, in seconds.

  data: i/o, R(*)*8
    Data vector for this frame.  Contains both unfiltered and
    filtered signals.

  reset: input, L*4
    Interval start flag.  This flag will be true on the first
    frame of each requested time interval; it will be false on
    all other frames.  This allows for the initialization of
    counters, integrators, etc.

DESCRIPTION
  This subroutine evaluates the filters.  It is called after
  allocateFilt, activateFilt and reMapFilt.  It is called one
  time for each record of each input file.

  DoFilt will need the compressed channel numbers placed in
  common block /FiltCom/ by subroutine reMapFilt.  It should
  perform the following operations.

  1. Use the table defined by allocateFilt to match the input
     file number to the appropriate filters.

  2. For each filter in the compressed list for that file, copy
     the data from the unfiltered channel to the filtered
     channel.  Separating this step from the actual filtering
     makes it easy to concenate filters as is often useful.  It
     is fairly common, for instance, to concatenate a lowpass
     and a notch filter on the same channels.  If the unfiltered
     data is first copied to the filtered channel, each the
     filter can then do its work in place, regardless of whether
     it is the first filter in the concatenation or not.
```

3. Then call subroutines to perform the appropriate recursive filtering in place. It is normally most flexible to have the actual filtering done in these subroutines one level lower rather than directly in subroutine doFilt.

EXAMPLES

A sample set of filter subroutines is in the files sample.filt.f and sample.filt.com in directory /user/maine/getData.3.1/source.

CAVEATS

The values in the filtered signal channels of data are not guaranteed to be retained between calls. If you need to save an output value between calls, you must save it in a local or common variable.

The question of what to do when a time dropout is detected (assuming that you test for such conditions at all) is complicated. I do not know a simple all-inclusive answer other than to suggest that the filtered data is likely to be questionable in the immediate vicinity of a dropout. There are probably ways of "properly" filtering through dropouts, but they are likely to be complicated.

ERROR HANDLING

There are no special provisions for error handling.

SEE ALSO

calculations [topic]
allocateFilt, activateFilt [sub]

KEYWORDS

doFilt subroutine,
do/perform/evaluate filtering/filters

AUTHOR Richard Maine - NASA Dryden
VERSION 3.1.1
DATE 8 Sept 86

A.6 File Read Subroutine Help Files

A.6.1 Function *OpenR*—

```
openR [sub] -- open a time history file for reading

USAGE
  logical = openR(unit,name,nChans)

PARAMETERS
  unit: input, I*4
    fortran unit number.

  name: input, C*(*)
    file name.

  nChans: output, I*4
    number of channels available on the file.

  openR: return, L*4
    true if open is successful.

DESCRIPTION
  Opens a time history data file for reading.  This is one of
  the time history file interface routines.  It must be called
  before any other reference to a file by the file interface
  read routines.  There are several different versions of the
  routine for accessing different file formats.  The interface
  to all versions is identical.

EXAMPLES
  integer unit,nChans
  logical openR
  if (.not.openR(unit,'data',nChans) ) write(*,*) 'open failed.'

CAVEATS
  Some versions may support only one time history data file open
  for reading at a time.

SEE ALSO
  fileInterface, unc1, unc2, cmp2, openW, closeW, fWrite
  closeR, rSigs, sigsR, chansR, rewR, fRead, fSeek
```

LIBRARY

Jobs using a particular set of file interface routines must bind in the library containing the appropriate routines. In addition, most of the interface libraries use subroutines in /user/maine/lib/misc.lib.o.

The interface subroutines are in the directory /user/maine/fRead.

The library /user/maine/fRead/auto.o is the most generally useful one. It automatically recognizes several file formats and reads them appropriately. It can also handle multiple files simultaneously opened with different formats.

KEYWORDS

openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
time history data file read/access/interface
subroutines/routines,
open a file for read

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/19/85

A.6.2 Subroutine *RSigs*—

```
rSigs [sub] -- return list of signal names on time history file

USAGE
  call rSigs(unit,sigs)

PARAMETERS
  unit: input, I*4
    fortran unit number.

  sigs: output, C(*)*(*)
    List of names of the available signals.  This list is in the
    order of the channels on the file.  It reflects all
    available signals, not the currently selected list of
    signals to be read.

DESCRIPTION
  This is one of the time history interface routines.  It
  returns a list of the names of the signals available on a
  file.  The list returned includes all available signals, in
  the order of their channel numbers; i.e., it is the list of
  signals that would be returned if sigsR or chansR were not
  called.  Any calls to rSigs must be after openR is called and
  before closeR is called for the referenced file.  Some
  implementations may further restrict rSigs to be illegal after
  any calls to fRead or fSeek for the referenced file.  To be
  compatable with all implementations, you should adhere to this
  restriction.

  There are several different versions of the routine for
  accessing different file formats.  The interface to all
  versions is identical.

EXAMPLES
  integer unit
  character sigs(200)*16
  call rSigs(unit,sigs)

SEE ALSO
  fileInterface, unc1, unc2, cmp2, openW, closeW, fWrite
  openR, closeR, sigsR, chansR, rewR, fRead, fSeek
```

LIBRARY

  Jobs using a particular set of file interface routines must bind in the library containing the appropriate routines. In addition, most of the interface libraries use subroutines in /user/maine/lib/misc.lib.o.

  The interface subroutines are in the directory /user/maine/fRead.

  The library /user/maine/fRead/auto.o is the most generally useful one. It automatically recognizes several file formats and reads them appropriately. It can also handle multiple files simultaneously opened with different formats.

KEYWORDS

  openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
  time history data file read/access/interface
  subroutines/routines,
  return/get list of available (data channel)/signal names

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/19/85

A.6.3 Subroutine *SigsR*—

```
sigsR [sub] -- specify signals to read from time history file

USAGE
  call sigsR(unit,sigs,nChans)

PARAMETERS
  unit: input, I*4
    fortran unit number.

  sigs: input, C(*)*(*)
    List of names of the signals to be read.  The data vector
    returned from subsequent calls to fRead or fSeek will
    contain values for these signals.  A signal name of " "
    (blank) indicates that a constant value of 0.  is to be
    returned to the corresponding location.  Signal names may be
    repeated in this list to duplicate values to 2 or more
    locations in the data vector.  Signal names are not case
    sensitive.

  nChans: input, I*4
    Length of the sigs vector; i.e., the number of signals to be
    read.  Currently limited to a maximum of 1000 (but this
    limit can easily be increased as needed).

DESCRIPTION
  This is one of the time history interface routines.  It
  specifies the signals to be read by subsequent calls to fRead
  or fSeek.  It can be called at any time after the initial call
  to openR for a file.  It takes effect immediately.  This
  routine can be called any number of times to change the
  signals being read.  When a file is opened by openR, it is
  initialized to return all of the available channels in
  numerical order; this order is in effect until the first call
  to chansR or sigsR.

  There are several different versions of the routine for
  accessing different file formats.  The interface to all
  versions is identical.

  Signals to be read can alternately be specified by calling
  chansR, which is simillar to sigsR, except that chansR finds
  signals by channel number instead of by name.
```

EXAMPLES

```
integer unit,nChans
character sigs(200)*16
call sigsR(unit,sigs,nChans)
```

ERROR HANDLING

In most versions, signal names not matching available signals result in an error message and return the constant value 0 in the corresponding data location.

SEE ALSO

fileInterface, uncl, unc2, cmp2, openW, closeW, fWrite
openR, closeR, rSigs, chansR, rewR, fRead, fSeek

LIBRARY

Jobs using a particular set of file interface routines must bind in the library containing the appropriate routines. In addition, most of the interface libraries use subroutines in /user/maine/lib/misc.lib.o.

The interface subroutines are in the directory /user/maine/fRead.

The library /user/maine/fRead/auto.o is the most generally useful one. It automatically recognizes several file formats and reads them appropriately. It can also handle multiple files simultaneously opened with different formats.

KEYWORDS

openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
time history data file read/access/interface
subroutines/routines,
specify/select (data channels)/signals for read

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/19/85

A.6.4 Subroutine *ChansR*—

```
chansR [sub] -- specify channels to read from time history file

USAGE
  call chansR(unit,chans,nChans)

PARAMETERS
  unit: input, I*4
    fortran unit number.

  chans: input, I(*)*4
    List of the channel numbers to be read.  The data vector
    returned from subsequent calls to fRead or fSeek will
    contain values for these channels.  Channel numbers must be
    between 0 and the number of channels available on the file
    (as returned by openR).  A channel number of 0 indicates
    that a constant value of 0.  is to be returned to the
    corresponding location.  Channel numbers may be repeated in
    this list to duplicate values to 2 or more locations in the
    data vector.

  nChans: input, I*4
    Length of the chans vector; i.e., the number of channels to
    be read.  Currently limited to a maximum of 1000 (but this
    limit can easily be increased as needed).

DESCRIPTION
  This is one of the time history interface routines.  It
  specifies the channels to be read by subsequent calls to fRead
  or fSeek.  It can be called at any time after the initial call
  to openR for a file.  It takes effect immediately.  This
  routine can be called any number of times to change the
  channels being read.  When a file is opened by openR, it is
  initialized to return all of the available channels in
  numerical order (i.e.  chans(i)=i); this order is in effect
  until the first call to chansR or sigsR.

  There are several different versions of the routine for
  accessing different file formats.  The interface to all
  versions is identical.

  Channels to be read can alternately be specified by calling
  sigsR, which is simillar to chansR, except that sigsR finds
  channels by name instead of by channel number.
```

EXAMPLES

```
integer unit,nChans,chans(200)
call chansR(unit,chans,nChans)
```

SEE ALSO

fileInterface, unc1, unc2, cmp2, openW, closeW, fWrite
openR, closeR, rSigs, sigsR, rewR, fRead, fSeek

LIBRARY

Jobs using a particular set of file interface routines must bind in the library containing the appropriate routines. In addition, most of the interface libraries use subroutines in /user/maine/lib/misc.lib.o.

The interface subroutines are in the directory /user/maine/fRead.

The library /user/maine/fRead/auto.o is the most generally useful one. It automatically recognizes several file formats and reads them appropriately. It can also handle multiple files simultaneously opened with different formats.

KEYWORDS

openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
time history data file read/access/interface
subroutines/routines,
specify/select (data channels)/signals for read

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/19/85

A.6.5 Subroutine *RewR*—

```
rewR [sub] -- rewind a time history data file

USAGE
  call rewR(unit)

PARAMETERS
  unit: input, I*4
    fortran unit number.

DESCRIPTION
  This is one of the time history interface routines.  It
  repositions an input time history file so that the next call
  to fRead will return the first record of the file.  It can be
  called at any time after the initial call to openR for a file.

  There are several different versions of the routine for
  accessing different file formats.  The interface to all
  versions is identical.

EXAMPLES
  integer unit
  call rewR(unit)

SEE ALSO
  fileInterface, unc1, unc2, cmp2, openW, closeW, fWrite
  openR, closeR, rSigs, chansR, sigsR, fRead, fSeek

LIBRARY
  Jobs using a particular set of file interface routines must
  bind in the library containing the appropriate routines.  In
  addition, most of the interface libraries use subroutines in
  /user/maine/lib/misc.lib.o

  The interface subroutines are in the directory
  /user/maine/fRead.

  The library /user/maine/fRead/auto.o is the most generally
  useful one.  It automatically recognizes several file formats
  and reads them appropriately.  It can also handle multiple
  files simultaneously opened with different formats.
```

KEYWORDS
openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
time history data file read/access/interface
subroutines/routines,
rewind/reposition a time history file

AUTHOR Richard Maine - NASA Dryden

VERSION 1.2
DATE 11/19/85

A.6.6 Function *FSeek*—

```
fSeek [sub] -- read a random record from a time history file

USAGE
  logical = fSeek(unit,tSeek,time,data)

PARAMETERS
  unit: input, I*4
    fortran unit number.

  tSeek: input, R*8
    time requested, seconds.

  time: output, R*8
    time of the record returned, seconds.

  data: output, R(*)*8
    data values for this time.  The values are in the order
    previously specified by calling sigsR or chansR, or in the
    default order for the file if neither sigsR nor chansR has
    been called.

  fSeek: return, L*4
    returns true if a record was successfully read.  If there
    was no data at or after the requested time, then fSeek
    returns false.  In this event, the values of time and data
    are undefined.

DESCRIPTION
  This is one of the time history interface routines.  It
  repositions a time history file to a requested time and
  returns a record of data.  It can be called at any time after
  the initial call to openR for a file.  The record returned is
  the first record with time greater than or equal to the
  requested time.  If there is no such record, then fSeek
  returns a false value.

  There are several different versions of the routine for
  accessing different file formats.  The interface to all
  versions is identical.

EXAMPLES
  integer unit
  logical fSeek
  double precision tSeek,time,data
  if (.not.fSeek(unit,tSeek,time,data)) write(*,*) 'no such time'
```

CAVEATS

A successful (true) return from fSeek is no guarantee that the returned time is anywhere near the requested time. It indicates only that the returned time is later. If the requested time is before the first available time or is during a time interval missing from the file, the actual time returned may be substantially later.

The intent of fSeek is to provide fast random access to the beginning of a time interval, with subsequent records to be retrieved by fRead. The implementation varies widely with different file types. With some file types, it is impractical to randomly reposition a file. In these cases, fSeek may be implemented by rewinding and then reading to the desired record. Therefore, truly random access to individual records should be avoided; it will work, but may be excruciatingly slow, depending on the file type.

SEE ALSO

fileInterface, unc1, unc2, cmp2, openW, closeW, fWrite
openR, closeR, rSigs, chansR, sigsR, rewR, fRead

LIBRARY

Jobs using a particular set of file interface routines must bind in the library containing the appropriate routines. In addition, most of the interface libraries use subroutines in /user/maine/lib/misc.lib.o.

The interface subroutines are in the directory /user/maine/fRead.

The library /user/maine/fRead/auto.o is the most generally useful one. It automatically recognizes several file formats and reads them appropriately. It can also handle multiple files simultaneously opened with different formats.

KEYWORDS

openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
time history data file read/access/interface
subroutines/routines,
read data/(random record) from a time history file,
reposition a time history file

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/19/85

A.6.7 Function *FRead*—

```
fRead [sub] -- read next record from a time history file

USAGE
  logical = fRead(unit,time,data)

PARAMETERS
  unit: input, I*4
    fortran unit number.

  time: output, R*8
    time of the record returned, seconds.

  data: output, R(*)*8
    data values for this time.  The values are in the order
    previously specified by calling sigsR or chansR, or in the
    default order for the file if neither sigsR nor chansR has
    been called.

  fRead: return, L*4
    returns true if a record was successfully read.  If there
    was no more data to read, then fRead returns false.  In this
    event, the values of time and data are undefined.

DESCRIPTION
  This is one of the time history interface routines.  It
  returns data from the next sequential record of a time history
  data file.  It can be called at any time after the initial
  call to openR for a file.  The initial call to openR
  initializes a file to return the first available record.
  Records are then returned in sequential order, except as
  modified by calls to fRew or fSeek.

  There are several different versions of the routine for
  accessing different file formats.  The interface to all
  versions is identical.

EXAMPLES
  integer unit
  logical fRead
  double precision time,data
  if (.not.fRead(unit,time,data)) write(*,*) 'no more data'
```

SEE ALSO
  fileInterface, unc1, unc2, cmp2, openW, closeW, fWrite
  openR, closeR, rSigs, chansR, sigsR, rewR, fSeek

LIBRARY
  Jobs using a particular set of file interface routines must
  bind in the library containing the appropriate routines. In
  addition, most of the interface libraries use subroutines in
  /user/maine/lib/misc.lib.o.

  The interface subroutines are in the directory
  /user/maine/fRead.

  The library /user/maine/fRead/auto.o is the most generally
  useful one. It automatically recognizes several file formats
  and reads them appropriately. It can also handle multiple
  files simultaneously opened with different formats.

KEYWORDS
  openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
  time history data file read/access/interface
  subroutines/routines,
  read data/(next record) from a time history file

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/19/85

A.6.8 Subroutine *CloseR*—

```
closeR [sub] -- close a time history data file

USAGE
  call closeR(unit)

PARAMETERS
  unit: input, I*4
    fortran unit number.

DESCRIPTION
  This is one of the time history interface routines.  It closes
  an input time history file.  It can be called at any time
  after the initial call to openR for a file.  If called for a
  file that has not been opened, it has no effect.  After closeR
  has been called, no more time history interface routines can
  be called for that unit until openR has been called again.  It
  is allowed to close a unit with closeR and then re-open the
  unit with openR for the same or a different file.

  You can usually get by without calling closeR if you will be
  making no more calls to the file interface routines.  Use of
  closeR is advisable, however, and may help avoid conflicts
  with other jobs.

  There are several different versions of the routine for
  accessing different file formats.  The interface to all
  versions is identical.

EXAMPLES
  integer unit
  call closeR(unit)

SEE ALSO
  fileInterface, unc1, unc2, cmp2, openW, closeW, fWrite
  openR, rSigs, chansR, sigsR, rewR, fRead, fSeek

LIBRARY
  Jobs using a particular set of file interface routines must
  bind in the library containing the appropriate routines.  In
  addition, most of the interface libraries use subroutines in
  /user/maine/lib/misc.lib.o
```

The interface subroutines are in the directory /user/maine/fRead.

The library /user/maine/fRead/auto.o is the most generally useful one. It automatically recognizes several file formats and reads them appropriately. It can also handle multiple files simultaneously opened with different formats.

KEYWORDS
openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
time history data file read/access/interface
subroutines/routines,
close a time history file

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/19/85

A.7 File Write Subroutine Help Files

A.7.1 Function *OpenW*—

```
openW [sub] -- create and open a new time history file

USAGE
  logical = openW(unit,name,nChans,sigs,format)

PARAMETERS
  unit: input, I*4
    fortran unit number.

  name: input, C*(*)
    file name.

  nChans: input, I*4
    number of channels to be written on the file.  Currently
    limited to 1000, but this limit can be easily changed.

  sigs: input, C(*)*(*)
    List of names of the signals to be written.  This list must
    be in the same order as the signals will be supplied to
    fWrite.  There are nChans elements of the list.  The signal
    names should be left-justified and contain no embedded
    blanks or other special characters.  The write routines will
    work with any signal names, but many programs that access
    the files will have trouble parsing their input if the
    signal names contain special characters.  The names are
    case-insensitive, so case can be freely used to enhance
    readability.  Duplicate signal names will cause problems in
    most programs and should be avoided.

  format: input, C*(*)
    Format to be used for the file.  The interpretation of this
    parameter depends on the particular write routines.  A
    particular set of write routines is free, for instance, to
    ignore this parameter and write in a single pre-determined
    format.  Alternatively, a set of write routines supporting a
    particular format can verify that the requested format is
    the supported one.  The auto write routines use this
    parameter to determine which of several supported formats to
    write.

  openW: return, L*4
    true if open successful.
```

DESCRIPTION

Creates and opens a new time history data file for writing. This is one of the time history file interface routines. It must be called before any other reference to a file by the file interface write routines. There are several different versions of the routine for creating different file formats. The interface to all versions is identical.

EXAMPLES

```
integer unit,nChans
character sigs(200)*16
logical openW
if (.not. openW(unit,'data',nChans,sigs,'cmp2')) then
  write (*,*) 'oops'
endif
```

CAVEATS

Most versions of OpenW attempt to delete any pre-existing file of the same name in order to avoid conflicting file structure data. Therefore, you can not use makeFile to override characteristics of a file about to be written. You can use file equates for such overrides. Equate specifications incompatable with the particular routines may cause various errors.

Poor signal name choices (such as names with embedded or leading blanks) will cause no problems when writing the file; it will just make access to those signals difficult for many programs.

SEE ALSO

fileInterface, unc1, unc2, cmp2, closeW, fWrite
openR, closeR, rSigs, sigsR, chansR, rewR, fRead, fSeek

LIBRARY

Jobs using a particular set of file interface routines must bind in the library containing the appropriate routines. In addition, most of the interface libraries use subroutines in /user/maine/lib/misc.lib.o.

The interface subroutines are in the directory /user/maine/fWrite. The most commonly used set is /user/maine/fWrite/auto.o.

```
KEYWORDS
  openW, closeW, fWrite,
  time history data file write/interface subroutines/routines,
  open a file for write

AUTHOR Richard Maine - NASA Dryden
VERSION 2.1
DATE 12/27/85
```

A.7.2 Subroutine *FWrite*—

```
fWrite [sub] -- write record to a time history file

USAGE
  call fWrite(unit,time,data)

PARAMETERS
  unit: input, I*4
    fortran unit number.

  time: input, R*8
    time of the record, seconds.

  data: input, R(*)*8
    data values for this time.  The values must be in the order
    specified in the openW call.  The number of data values must
    agree with the number specified in the call to openW.

DESCRIPTION
  This is one of the time history interface routines.  It writes
  data to the next sequential record of a time history data
  file.  OpenW must previously have been called to open the file
  for writing.  Subroutine fWrite is then called repeatedly to
  write the records on the file.  This must be followed by a
  call to closeW to close the file.

  There are several different versions of the routine for
  accessing different file formats.  The interface to all
  versions is identical.

EXAMPLES
  integer unit
  double precision time,data
  call fWrite(unit,time,data)

CAVEATS
  The times in successive calls to fWrite are assumed to be in
  increasing order.  There is no provision for fWrite to sort
  the records internally.  This should be enforced by the
  calling program.  The consequences of violating this
  limitation may vary widely depending on the particular
  implementation.  Some implementations may abort.  Other
  implementations may write a file that can be read
  sequentially, but cannot be positioned with fSeek.  Some
  implementations may even work (but none of the current ones
  do).
```

SEE ALSO

fileInterface, unc1, unc2, cmp2, openW, closeW
openR, closeR, rSigs, chansR, sigsR, rewR, fRead, fSeek

LIBRARY

Jobs using a particular set of file interface routines must bind in the library containing the appropriate routines. In addition, most of the interface libraries use subroutines in /user/maine/lib/misc.lib.o.

The interface subroutines are in the directory /user/maine/fWrite. The most commonly used set is /user/maine/fWrite/auto.o.

KEYWORDS

openW,closeW,fWrite,
time history data file write/interface subroutines/routines,
write data/(next record) to a time history file

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/19/85

A.7.3 Subroutine *CloseW*—

```
closeW [sub] -- close a time history data file

USAGE
  call closeW(unit)

PARAMETERS
  unit: input, I*4
    fortran unit number.

DESCRIPTION
  This is one of the time history interface routines.  It closes
  an output time history file.  It can be called at any time
  after the initial call to openW for a file.  If called for a
  file that has not been opened, it has no effect.  After closeW
  has been called, no more time history interface routines can
  be called for that unit until openR is called to open it for
  reading.  It is not allowed to re-open the file for writing;
  any such attempt will delete the old data and create a new
  file.

  You must call closeW in order to finish the creation of a time
  history file.  If closeW is not called, the resulting file may
  be missing critical information required for the read routines
  to work.

  There are several different versions of the routine for
  accessing different file formats.  The interface to all
  versions is identical.

EXAMPLES
  integer unit
  call closeW(unit)

SEE ALSO
  fileInterface, unc1, unc2, cmp2, openW, fWrite
  openR, closeR, rSigs, chansR, sigsR, rewR, fRead, fSeek

LIBRARY
  Jobs using a particular set of file interface routines must
  bind in the library containing the appropriate routines.  In
  addition, most of the interface libraries use subroutines in
  /user/maine/lib/misc.lib.o.
```

The interface subroutines are in the directory
/user/maine/fWrite. The most commonly used set is
/user/maine/fWrite/auto.o.

KEYWORDS
openW,closeW,fWrite,
time history data file write/interface subroutines/routines,
close a time history file

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/19/85

A.8 File Format Help Files

A.8.1 ASCII 1 Format—

```
asc1 [file] -- ascii 1 file format

DESCRIPTION
  This is a simple ascii format intended primarily for transfer
  of data tapes between different computers.  It is not
  recommended for internal Elxsi use because of its
  inefficiency, both in file size and access time.

EXAMPLES
  A short sample file is in /user/maine/helpFiles/file/asc1.sample.

TAPE SPECIFICATIONS
  As the format is primarily aimed at tape data transfer, this
  section documents preferred tape characteristics.  The format
  is not actually limited to tape media.

    9-track tape
    6250 bpi preferred, 1600 bpi available, limited 800 bpi
      capability.
    ANSI labeled preferred. Unlabelled available if needed.
    ASCII coded data, parity bit is always 0.
    Fixed length 8000-character blocks.  Last block in a file
      may be shorter.  Other block lengths are available if
      needed, subject to the restrictions that the length must
      be a multiple of 80 and must be no more than 32720.

RECORD STRUCTURE
  The data is organized into fixed length 80-character records.
  In most cases, a logical record requires more than 80
  characters; the logical record is then split into multiple
  80-character records.  Any unused fields in a record are
  padded with blanks.

HEADER RECORDS
  The first several records on a file are header records
  describing what signals are on the tape.  The first 8
  characters of each header record are a tag to identify the
  type of data on that record.  As currently implemented, these
  tags are redundant, because the exact same records are always
  written in exactly the same order.  The format does allow for
  future expansion by the addition of more header records and
  programs accessing the files should take this into account; at
  a minimum the programs should verify that the header records
  found agree with those expected.
```

All character data in the header records, including the record type tags and the signal names, should be treated in a case-insensitive manner on machines that distinguish between upper and lower case letters. All character data are left-justified in their fields. Character constant values are indicated below in quotations. The quote marks are not actually part of the data.

A. format record.

The first record of the file identifies the file format. This makes provision for automatic handling of different formats. The fortran format of the record is (a8,a8).

Columns	Field-Name	Field-Format	Value
1-8	record-Type	a8	constant - 'format'
9-16	file-format	a8	constant - 'asc 1'

B. nChans record.

The second record of the file specifies the number of channels (signals) contained on the file. The fortran format of the record is (a8,i8).

Columns	Field-Name	Field-Format	Value
1-8	record-Type	a8	constant - 'nChans'
9-16	nChans	i8	variable - number of chans

C. names records.

The 3rd logical record of the file spcifies the names of the signals on the file. This logical record is continued across as many physical 80-column records as required. The format of the continuation records, if any, is slightly different from that of the initial record. (Fortran naturally handles this with the format shown here). The fortran format of the initial record is (a8,8x,4a16), and that of the continuation records is (5a16); the entire logical record is naturally read with the fortran format (a8,8x,4a16/(5a16)).

The format allows names up to 16 characters long. Particular projects are likely to restrict the names actually used to smaller limits in order to accomodate programs unable to handle longer names. Note that shorter names are always left justified in the 16 character fields.

Initial record layout:

Columns	Field-Name	Field-Format	Value
1-8	record-Type	a8	constant - 'names'
9-16	unused	8x	
17-32	name-1	a16	variable - name of sig 1
33-48	name-2	a16	variable - name of sig 2
49-64	name-3	a16	variable - name of sig 3
65-80	name-4	a16	variable - name of sig 4

Continuation records contain 5 names each (possibly less on the last record) in 16-character fields.

D. data001 record.

The data001 record indicates the end of the header records. The purpose of this record is to allow for easy future expansion of the header records. The preferred way to position the file at the beginning of the actual data is to rewind and search for the data001 record. Programs using this method will work unchanged if future additional header records are defined (assuming that the programs do not need the information in the new headers).

Columns	Field-Name	Field-Format	Value
1-8	record-Type	a8	constant - 'data001'

DATA RECORDS

The remainder of the file, after the header records, consists of data records. The data for each time consititutes a single logical data record. This logical record can (and usually does) span several physical 80-character records.

For each time, there is a single value for every signal on the file. There is no provision for data compression or for multi-sample-rate data on a single file. If a signal was sampled at a higher rate than the sample rate of the file, then the signal will be thinned. If a signal was sampled at a lower rate than the sample rate of the file, each sample will be repeated multiple times on the file. (Note that this is hold-last-value processing, NOT linear interpolation). If precise representation of data at multiple sample rates is needed, then the data at each sample rate must be requested as a separate file.

Although each file will have a nominal sample rate, it is not guaranteed to be an absolutely fixed rate. There may be time dropouts. Also, if the PCM system is not running at exactly the nominal rate, the processing will follow the PCM system, not the nominal rate. The time of each record is indicated in the first field of the record. This time is accurate. Times implied by assuming exactly constant nominal sample rates are not guaranteed to be accurate.

All data are in format g20.14, 4 fields to a physical record. (The time is actually written in format (f10.3,10x), but this can be read as a g20.14 field with no special fortran considerations).

The data in the data records are time, followed by the data values. Time and the first 3 data values are on the first 80-column record of each time point; the following records for each time point have 4 data values each (possible less on the last record of a time point). The data values are in the same order as listed in the names header record.

Time is in floating point seconds past midnight (usually local time, but this may vary from project to project).

All data values are represented as floating point engineering units values. The units of measure for each signal are separately documented. Any integer values (such as digital words) are converted to floating point for consistency. Character-valued data is not supported.

POSSIBLE VARIATIONS AND FUTURE PLANS

The 20 character data fields are quite liberal to ensure that no accuracy will be lost. They do, however, require quite a bit more tape than smaller field widths. We will consider requests for formats with smaller fields that use less file size at the cost of some accuracy. Field widths as small as 10 characters may be acceptable for some applications, but the accuracy may be marginal (only 4 significant digits can be guaranteed to fit in a 10 character field with standard fortran formats). It would make the format considerably more complicated to mix field lengths in the same file, and we do not propose to do that.

Future expansion may include the addition of additional header records giving such data as time skew and units of measure. Guidelines are given above for how to program in a way guaranteed to be compatable with such future expansions.

SUBROUTINES

For read access, use /user/maine/fRead/auto.O, which automatically recognizes this or several other formats. A set of write routines that handles this and other formats is in /user/maine/fWrite/auto.o. Both read and write routines are currently limited to 10 simultaneously open files.

CAVEATS

Current dimensions allow up to 1000 channels per file. This can be easily changed if required.

SEE ALSO

fileInterface,openR,closeR,rSigs,sigsR,chansR,rewR,fread,fSeek, openW,closeW,fWrite

KEYWORDS

ascl/ascii file format/access,
time history data file read/write/access/interface
subroutines/routines,
openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
openW, closeW, fWrite

AUTHOR Richard Maine - NASA Dryden
VERSION 1.1
DATE 1/8/86

A.8.2 Compressed 2 Format—

```
cmp2 [file] -- compressed file format 2

DESCRIPTION
  This is a compressed format.  It uses byte-aligned R*4 data to
  make access relatively fast and easy.  There are header
  records describing various aspects of the file and its data.
  The format is designed primarily for KAM access, but is
  largely compatable with sequential access.  Early versions may
  be sequential until KAM matures sufficiently.

  The cmp1 format (now obsolete) is identical to cmp2, except
  that cmp1 omits the 'data001' header record.

  DATA RECORD FORMAT
    time: I*4 -- This is the primary record key.  It is scaled
      time.  The actual time in secs is
      time0+timeScale*(time-keyOffset), where keyOffset, time0
      and timeScale are specified in the header.
    recFlags: I*1 -- Record type flags.
      Bits 6-7 (lsb) are 00 for a full frame, 01 for a bit-map
        compressed frame, or 10 for a channel-list frame.  The
        value 11 is reserved for future enhancements.
      The other 6 bits are currently unused.
    chanFlags: Bit(nChans) -- This field is used iff recFlags
      is 1.  When this field is present, each bit represents a
      channel.  A 1 means this record has a value for that
      channel; 0 means the previous value should be retained.
      The field is padded with 0's to the next byte boundary.
    chanList: I(var)*1 -- This field is used iff recFlags is 2.
      When the field is present, each byte is an unsigned
      channel number, indicating that the record has a data
      value for that channel.  The list is terminated by a zero
      byte.
    data: R(nChans)*3 -- Data values.  There is one data value
      for each channel specified by the chanFlags or chanList.
      Full-frame records have a value for every channel.  An
      R*3 value is just an R*4 value, with the low-order
      byte omitted.

  HEADER RECORD FORMATS
    Header records have primary keys 0<primaryKey<keyOffset,
    where keyOffset is specified in the header.  They also have
    an 8-character descriptive secondary key, which is the
    second field of the record.  Secondary keys are not required
    to be unique.
  Key=1,'format' +req
    format: c*8 = 'cmp 2'
    version: c*8 = '.1'
  Key=100,'headers' +req
    dummy: I*4 -- currently unused.  hardwired to 1000000.
    lastHeaderKey: I*4 -- key of the last defined header record.
    headerKeySpace: -- key spacing for header records.

  Key=200,'timeKey'
    baseTime,timeScale: R*8 (currently=0.,2**-12)
    keyOffset: I*4 (currently=2**20)
    fullInterval: I*4 -- full frame interval in key units
                         (currently=10240)
  Key=300,'nChans' +req
    nChans: I*4 -- number of channels
  Key=400,410,420,430,'names1','names2','names3','names4' +req
    names: C(nChan)*nameLen.
    -- These 4 records contain the signal names, spilt into 4
       parts.  The first 4 characters of each name are in the
       'names1' record, the second 4 characters of each name are
       in 'names2' record, etc.  The names are 16 characters
       long.  (The strange splitting of the names into
       4-character chunks is to prevent this record from
       quadrupling the maximum record size needed for the file,
       which could adversely impact storage efficiency).
  Key=?,'?'
    user-specified data.  (unimplemented)
```

SUBROUTINES

For read access, use /user/maine/fRead/auto.O, which automatically recognizes this or several other formats. A set of write routines that handles this and other formats is in /user/maine/fWrite/auto.o. Both read and write routines are currently limited to 10 simultaneously open files.

CAVEATS

Current dimensions allow up to 1000 channels per file. This can be easily changed if required.

SEE ALSO

fileInterface,openR,closeR,rSigs,sigsR,chansR,rewR,fread,fSeek,
openW,closeW,fWrite

IMPLEMENTATION

Currently uses SAM, which makes random access slow. KAM versions have been tested, but not released for general use.

```
KEYWORDS
  cmp2/cmp1/compressed file format/access,
  time history data file read/write/access/interface
  subroutines/routines,
  openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
  openW, closeW, sigsW, fWrite

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 11/7/85
```

A.8.3 List 1 Format—

```
lis1 [file] -- list 1 file format

DESCRIPTION
  This is a simple Ascii format intended primarily for listing
  to terminal screens or printers.  Only writing of this format
  is supported; files in this format are intended for human
  examination, not input to computer programs.  For Ascii file
  transfer, use asc1 format instead.

  This format puts up to 5 data values per line.  The data
  values are formatted with five digits of precision.  Time is
  displayed in hours, minutes, seconds and milliseconds.

EXAMPLES
  A short sample file is in /user/maine/helpFiles/file/lis1.sample.

SUBROUTINES
  A set of of write routines that handles this and other formats
  is in /user/maine/fWrite/auto.o.  Read access to this format
  is not supported.

CAVEATS
  Current dimensions allow up to 1000 channels per file.  This
  can be easily changed if required.

SEE ALSO
  fileInterface,openR,closeR,rSigs,sigsR,chansR,rewR,fread,fSeek,
  openW,closeW,fWrite

KEYWORDS
  lis1/list file format/access,
  time history data file read/write/access/interface
  subroutines/routines,
  openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
  openW, closeW, fWrite

AUTHOR Richard Maine - NASA Dryden
VERSION 1.1
DATE 1/2/87
```

A.8.4 Uncompressed 1 Format—

unc1 [file] -- uncompressed file format 1

DESCRIPTION

This is a simple, uncompressed file format. All records are identical data records; there are no header records. The first item of each record is time, stored as R*8 seconds. Each data channel then has an R*4 value (converted to/from R*8 by the file access routines). This is a relatively close analog to 'mmle' format files as used on the CDC. There are no signal names or associated data.

Support for this format is limited and intended primarily for compatability with old files. Support may be further limited in the future. In particular, the automatic file format recognition may be disabled for this format. (The requirement to recognize this format degrades the error handling capabilities of the automatic recognition routines). This would require the user to bind special routines for reading this format.

USAGE

Signal names in sigsR and rSigs are taken to be the channel numbers, converted to left-justified character strings. Subroutine sigsW does nothing.

SUBROUTINES

Source code for write routines is in /user/maine/fWrite/unc1.f. I am not maintaining object code for writing in this format. For read access, use /user/maine/fRead/auto.o, which automatically recognizes this or several other formats. Source code for a less versatile and efficient, but more portable, set of read access routines is in /user/maine/fread.simple.f.

CAVEATS

See the paragraph in the description section warning of the limited support and possible future changes in the support of this format.

Current dimensions allow up to 1000 channels per file. This can be easily changed if required.

SEE ALSO
  fileInterface,openR,closeR,rSigs,sigsR,chansR,rewR,fread,fSeek,
  openW,closeW,fWrite

IMPLEMENTATION
  Straightforward, except for fSeek. The simple version does fSeek by rewinding and reading until the desired time (slow, but portable). The Elxsi-specific version operates similarly unless the records are of fixed-length type (which it determines by calling an Elxsi file system intrinsic). If the records are of fixed-length type, fSeek does a fast search for the start time using random access. The fWrite routines write fixed-length record types by default.

KEYWORDS
  uncl/mmle/uncompressed/fixed file format/access,
  time history data file read/write/access/interface
  subroutines/routines,
  openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
  openW, closeW, sigsW, fWrite

AUTHOR Richard Maine - NASA Dryden
VERSION 1.1
DATE 12/7/84

A.8.5 Uncompressed 2 Format—

```
unc2 [file] -- uncompressed file format 2

DESCRIPTION
  This is an uncompressed file format.  There are header records
  describing various aspects of the file and its data.  The file
  is designed for efficient access using Elxsi fortran
  extensions, which allow intermixed direct and sequential
  access.  The format is not inherently Elxsi-specific, however.
  A few convolutions in the format are to keep the header
  records the same length as the data records in order to allow
  simple direct access.

  DATA RECORD FORMAT
    The first item of each record is time, stored as R*8
    seconds.  Each data channel then has an R*4 value (converted
    to/from R*8 by the file access routines).  This is the same
    data record format as used in 'unc1' format files.

  HEADER RECORD FORMATS
    The first item in each header record is an 8-character
    descriptive key.  These keys need not be unique.  Occurances
    of multiple records with the same key mean that the data are
    concatenated to give the full fields.

  Key='format' +req (must be first record)
    format: c*8 = 'unc 2   '
    version: c*8 = '.1'
  Key='nChans' +req (must be second record)
    nChans: I*4 -- number of channels
  Key='title'
    title: C*(4*nChans) -- file title ('file title')
  Key='names' +req (currently hard-wired to recs 4-7)
    names: C(nChans)*4 -- channel names.
  Key='times001' +req
    sTime,eTime: R*8 -- interval start and end times
                        (unimplemented)
    sRec,eRec: I*4 -- interval start and end record numbers
                      (unimplemented)
  Key='data001' +req
    iTitle: C*(4*nChans) -- interval title ('interval 1')
    -- This record indicates the start of the data.  It must be
       the last record in the header portion of the file.
```

SUBROUTINES

For read access, use /user/maine/fRead/auto.O, which automatically recognizes this or several other formats. A set of write routines that handles this and other formats is in /user/maine/fWrite/auto.o. Both read and write routines are currently limited to 10 simultaneously open files.

Source code for a less versatile, but more portable set of read routines is in /user/maine/fRead/unc2.f. Source code for a portable set of write routines handling only this format is in /user/maine/fWrite/unc2.f.

CAVEATS

Current dimensions allow up to 1000 channels per file. This can be easily changed if required.

Current implementation supports only 1 interval per file.

SEE ALSO

fileInterface,openR,closeR,rSigs,sigsR,chansR,rewR,fread,fSeek, openW,closeW,fWrite

IMPLEMENTATION

Most of the routines are identical to their uncl format counterparts. The only difference is in the treatment of the header records. (Skipping over them after rewinds, etc.)

FUTURE PLANS

The hard-wired header record numbers should be removed, and key searches used instead. Also, provision should be made for other, user-specified header records. Treatment of multiple intervals in a file should be considered. Start-stop times and records should be filled in the times records.

KEYWORDS

unc2/uncompressed/fixed file format/access,
time history data file read/write/access/interface
subroutines/routines,
openR,closeR,rSigs,sigsR,chansR,rewR,fRead,fSeek,
openW, closeW, fWrite

AUTHOR Richard Maine - NASA Dryden
VERSION 1.2
DATE 1/8/86

Appendix B—Sample Calculation Routines

B.1 Sample Calculated Function Module

B.1.1 Subroutine *AllocateCF1*—

```
      subroutine allocateCF1

c Richard Maine.  12 Aug 86.
c Locate input and output signals for calculated function.
c Simple sample version for aileron, elevator and keas calculations.

      implicit none

c******************** common.
      common /CF1/ useDe,useDa,useKeas,iDeR,iDeL,oDe,oDa,iQbar,oKeas
        integer iDeR,iDeL,oDe,oDa,iQbar,oKeas
        logical useDe,useDa,useKeas
        save /CF1/

c******************** externals.
      external labelCalc,sigChan,calcChan,cantCalc
      integer sigChan,calcChan

c------------------------- executable code -------------------------

      call labelCalc(1,'CF1 sample.  Richard Maine  12 Aug 86')

c******************** locate input signals.
      iDeR = sigChan('der')
      iDeL = sigChan('del')
      iQbar = sigChan('qbar')

c******************** allocate calculated signals.
      oDe = calcChan('de')
      oDa = calcChan('da')
      okeas = calcChan('keas')
```

```
c******************** disable calculations needing unavailable signals.

c********** elevator and aileron calculations.
      if (iDeR.eq.0 .or. iDeL.eq.0) then
        call cantCalc(oDe)
        call cantCalc(oDa)
      endif

c********** keas calculation.
      if (iQbar.eq.0) call cantCalc(oKeas)

      return
      end
```

B.1.2 Subroutine *ActivateCF1*—

```
      subroutine activateCF1

c Richard Maine.  12 Aug 86.
c Activate needed calculated functions and their inputs.
c Simple sample version for aileron, elevator and keas calculations.

      implicit none

c********************* common.
      common /CF1/ useDe,useDa,useKeas,iDeR,iDeL,oDe,oDa,iQbar,oKeas
        integer iDeR,iDeL,oDe,oDa,iQbar,oKeas
        logical useDe,useDa,useKeas
        save /CF1/

c********************* externals.
      external isUsed,setUsed
      logical isUsed

c-------------------------- executable code --------------------------

c********************* de and da calculations.
      useDe = isUsed(oDe)
      useDa = isUsed(oDa)
      if (useDe .or. useDa) then
        call setUsed(iDeR)
        call setUsed(iDeL)
      endif

c********************* keas calculation.
      useKeas = isUsed(oKeas)
      if (useKeas) call setUsed(iQbar)

      return
      end
```

B.1.3 Subroutine *DoCF1*—

```
      subroutine doCF1 (time,data,reset)

c Richard Maine.  12 Aug 86.
c Evaluate calculated functions for getData.
c Simple sample version for aileron, elevator and keas calculations.

      implicit none

c********************* interface.
c time(input): time of this frame.
c data(i/o): data vector for both input and output.
c reset(input): true on the first point of a time segment.

      logical reset
      double precision time,data(0:*)

c********************* common.
      common /CF1/ useDe,useDa,useKeas,iDeR,iDeL,oDe,oDa,iQbar,oKeas
        integer iDeR,iDeL,oDe,oDa,iQbar,oKeas
        logical useDe,useDa,useKeas
        save /CF1/

c********************* external.
      intrinsic sqrt,max

c********************* local.
      double precision zero
      parameter (zero=0.)

c--------------------------- executable code ---------------------------

c***** de is the average of the left and right surfaces.
      if (useDe) data(oDe) = .5*(data(iDeR)+data(iDeL))

c***** da is (left-right)/2
      if (useDa) data(oDa) = .5*(data(iDeL)-data(iDeR))

c***** Keas
      if (useKeas) data(oKeas) = 17.17*sqrt(max(data(iQbar),zero))

      return
      end
```

B.2 Sample Filter Module

B.2.1 Subroutine *AllocateFilt*—

```
      subroutine allocateFilt (inF,inSig,nIn,maxOut,nOut)

c Richard Maine.  12 Aug 86.
c Locate input and output signals for filter.
c Sample version based on x-29.

      implicit none
      integer input,output
      parameter (input=5,output=6)

c******************** interface.
c inF(input): input file number.
c inSig(i/o): names of available input signals.
c             Output names appended on return.
c nIn(input): number of available input signals.
c maxOut(input): maximum allowed number of filtered signals.
c nOut(output): number of filtered signals.

      integer inF,nIn,maxOut,nOut
      character inSig(nIn)*(*)

c******************** common.
      integer maxInF,maxIch
      parameter (maxInF=10,maxIch=1000)
      common /filtCom/ iFilts,nFChs,nFChsU,fCh,fiCh,fChU,fiChU
      integer maxFF,maxFCh
      parameter (maxFF=2,maxFCh=50)
      integer iFilts(maxInF),nFChs(maxFF),nFChsU(maxFF),
     1  fCh(maxFCh,maxFF),fiCh(maxFCh,maxFF),fChU(maxFCh,maxFF),
     2  fiChU(maxFCh,maxFF)
      save /filtCom/

c******************** external.
      external sIndex
      integer sIndex
      intrinsic index,len
```

```
c******************** local.
      integer iFilt,i,iChan,iBlank

c nFF: number of files with filters.
c nFSigs: number of signals in each file with filters defined.
c fSigs: names of signals with filters defined.
      integer nFF,nFSigs(maxFF)
      character fSigs(maxFCh,maxFF)*16
      save nFF,nFSigs,fSigs

      data nFF/2/
      data nFSigs(1)/21/,(fSigs(i,1),i=1,21)/
     1  'al51010','aa52001','aa52002','aa52003',
     2  'va62002','va62004','va62005','va62006','da81001','da81002',
     3  'da81004','da81011','da81012','da81013','da81014',
     4  'da81015','da81016','da81018','da81019','da81021','da81022'/
      data nFSigs(2)/6/,(fSigs(i,2),i=1,6)/
     1  'al51012','da81030','al51001','al51007','al51008','al51009'/

c--------------------------- executable code ---------------------------

      nOut = 0

c******************** Find filter number for this file.
c******************** Based on first filtered signal.
c******************** Implementation allows only 1 filter per file.
      do 500 iFilt = 1 , nFF
        if (nFSigs(iFilt).gt.0) then
          if (sIndex(fSigs(1,iFilt),inSig,nIn).ne.0) goto 900
        endif
  500 continue
      iFilts(inF) = 0
      goto 9999
  900 iFilts(inF) = iFilt

c******************** find channel numbers of filtered signals.
      nFChs(iFilt) = 0
      do 2000 i = 1 , nFSigs(iFilt)
        iChan = sIndex(fSigs(i,iFilt),inSig,nIn)
        if (iChan.eq.0) then
          write (output,*) '*** filter source signal ',
     1      fSigs(i,iFilt),' not found.  Filter omitted.'
        else
          if (nOut.ge.maxOut) then
            write (output,*) '*** too many filtered signals. ',
     1        'List truncated'
            goto 9999
          endif
          nOut = nOut + 1
          nFChs(iFilt) = nOut
          fiCh(nOut,iFilt) = iChan
          fCh(nOut,iFilt) = nIn + nOut
          iBlank = index(inSig(iChan),' ')
          if (iBlank.lt.2 .or. iBlank.gt.len(inSig(1))-2)
     1      iBlank = len(inSig(1)) - 2
          inSig(nIn+nOut) = inSig(iChan)(1:iBlank-1) // '-f'
        endif
2000 continue
9999 return
     end
```

B.2.2 Subroutine *ActivateFilt*—

```
      subroutine activateFilt (inF,iaOff)

c Richard Maine.  9 Sept 86.
c Activate needed filters and their inputs.
c Sample version based on x-29.

      implicit none

c********************* interface.
c inF(input): input file number.
c iaOff(input): offset of channel numbers into allDat vector.

      integer inF,iaOff

c********************* common.
      integer maxInF,maxIch
      parameter (maxInF=10,maxIch=1000)
      common /filtCom/ iFilts,nFChs,nFChsU,fCh,fiCh,fChU,fiChU
      integer maxFF,maxFCh
      parameter (maxFF=2,maxFCh=50)
      integer iFilts(maxInF),nFChs(maxFF),nFChsU(maxFF),
     1  fCh(maxFCh,maxFF),fiCh(maxFCh,maxFF),fChU(maxFCh,maxFF),
     2  fiChU(maxFCh,maxFF)
      save /filtCom/

c********************* external.
      external isUsed,setUsed
      logical isUsed

c********************* local.
      integer i,iFilt
      logical useFilt

c-------------------------- executable code --------------------------

      iFilt = iFilts(inF)
      if (iFilt.ne.0) then

c********************* activate used filters and mark active inputs.
        do 1000 i = 1 , nFChs(iFilt)
          useFilt = isUsed(iaOff+fCh(i,iFilt))
          if (useFilt) call setUsed(iaOff+fiCh(i,iFilt))
 1000   continue
      endif
      return
      end
```

B.2.3 Subroutine *ReMapFilt*—

```
      subroutine reMapFilt (inF,iuMap)

c Richard Maine.  2 Sept 86.
c Remap filters to compressed locations.
c Sample version based on x-29.

      implicit none

c********************* interface.
c inF(input): input file number.
c iuMap(input): map from uncompressed to compressed locations.

      integer inF,iuMap(*)

c********************* common.
      integer maxInF,maxIch
      parameter (maxInF=10,maxIch=1000)
      common /filtCom/ iFilts,nFChs,nFChsU,fCh,fiCh,fChU,fiChU
      integer maxFF,maxFCh
      parameter (maxFF=2,maxFCh=50)
      integer iFilts(maxInF),nFChs(maxFF),nFChsU(maxFF),
     1  fCh(maxFCh,maxFF),fiCh(maxFCh,maxFF),fChU(maxFCh,maxFF),
     2  fiChU(maxFCh,maxFF)
      save /filtCom/

c********************* local.
      integer i,iu,iFilt

c--------------------------- executable code ---------------------------

      iFilt = iFilts(inF)
      if (iFilt.ne.0) then
        iu = 0
        do 1000 i = 1 , nFChs(iFilt)
          if (iuMap(fCh(i,iFilt)).ne.0) then
            iu = iu + 1
            fChU(iu,iFilt) = iuMap(fCh(i,iFilt))
            fiChU(iu,iFilt) = iuMap(fiCh(i,iFilt))
          endif
 1000   continue
        nFChsU(iFilt) = iu
      endif
      return
      end
```

B.2.4 Subroutine *DoFilt*—

```
      subroutine doFilt (inF,time,data,reset)

c Richard Maine.  12 Aug 86.
c calculate filtered data for an input record.
c Sample version based on x-29.

      implicit none

c********************* interface.
c inF(input): input file number.
c time(input): time of the record.
c data (i/o): data vector for both input and output.
c reset(input): forces the filter to be (re)initialized.

      integer inF
      logical reset
      double precision time,data(*)

c********************* common.
      integer maxInF,maxIch
      parameter (maxInF=10,maxIch=1000)
      common /filtCom/ iFilts,nFChs,nFChsU,fCh,fiCh,fChU,fiChU
      integer maxFF,maxFCh
      parameter (maxFF=2,maxFCh=50)
      integer iFilts(maxInF),nFChs(maxFF),nFChsU(maxFF),
     1  fCh(maxFCh,maxFF),fiCh(maxFCh,maxFF),fChU(maxFCh,maxFF),
     2  fiChU(maxFCh,maxFF)
      save /filtCom/

c********************* external.
      external low3F,notchF

c********************* local.
      integer i,iFilt
```

```
c--------------------------- executable code ----------------------------

      iFilt = iFilts(inF)
      if (iFilt.ne.0) then
        do 1000 i = 1 , nFChsU(iFilt)
          data(fChU(i,iFilt)) = data(fiChU(i,iFilt))
 1000   continue
        call low3F(iFilt,time,data,reset)
        call notchF(iFilt,time,data,reset)
      endif
      return
      end
```

B.2.5 Subroutine *Low3F*—

```
      subroutine low3F (iFilt,aTime,data,reset)

c Richard Maine.  12 Aug 86.
c 3rd order low-pass filter. (Really a concatenated 1st and 2nd order).
c u0,z0 are current in,out; u1,z1 previous time; z2 two previous.
c y0,y1,y2 are current, previous, and two previous intermediate state.
c Sample version based on x-29.

      implicit none

c******************** interface.
c iFilt(input): filter number.
c aTime(input): actual frame time.
c data (i/o): data vector for both input and output.
c reset(input): should filter be (re)initialized.

      integer iFilt
      logical reset
      double precision aTime,data(*)

c******************** common.
      integer maxInF,maxIch
      parameter (maxInF=10,maxIch=1000)
      common /filtCom/ iFilts,nFChs,nFChsU,fCh,fiCh,fChU,fiChU
      integer maxFF,maxFCh
      parameter (maxFF=2,maxFCh=50)
      integer iFilts(maxInF),nFChs(maxFF),nFChsU(maxFF),
     1  fCh(maxFCh,maxFF),fiCh(maxFCh,maxFF),fChU(maxFCh,maxFF),
     2  fiChU(maxFCh,maxFF)
      save /filtCom/

c******************** externals.
      intrinsic exp,cos

c******************** local.
c----- set appropriate break frequency.
      double precision freq
      parameter (freq=15.)
      integer iChan,i
      double precision wDt,dt(maxFF),
     1  eat(maxFF),eabt,c1(maxFF),c2(maxFF),g1(maxFF),
     2  g2(maxFF),u0i,y0i,z0i,
     3  u1(maxFCh,maxFF),y1(maxFCh,maxFF),y2(maxFCh,maxFF),
     4  z1(maxFCh,maxFF),z2(maxFCh,maxFF)
      save dt,eat,g1,c1,c2,g2,u1,y1,y2,z1,z2

      data dt/.005,.01/

c--------------------------- executable code ---------------------------

c********************* initialize filter at maneuver start.
      if (reset) then
c********** compute filter coefficients.
        wDt = freq*dt(iFilt)*2.*3.14159265
        eat(iFilt) = exp(-wDt)
        g1(iFilt) = .5*(1.-eat(iFilt))
        eabt = exp(-.866025404*wDt)
        c1(iFilt) = -2.*eabt*cos(.5*wDt)
        c2(iFilt) = eabt**2
        g2(iFilt) = .25*(1.+c1(iFilt)+c2(iFilt))
c********** initialize filter states.
        do 2000 i = 1 , nFChsU(iFilt)
          u0i = data(fChU(i,iFilt))
          u1(i,iFilt) = u0i
          y2(i,iFilt) = u0i
          y1(i,iFilt) = u0i
          z2(i,iFilt) = u0i
          z1(i,iFilt) = u0i
 2000   continue

c********************* filter.
      else
        do 4000 i = 1 , nFChsU(iFilt)
          iChan = fChU(i,iFilt)
          u0i = data(iChan)
          y0i = eat(iFilt)*y1(i,iFilt) + g1(iFilt)*(u0i+u1(i,iFilt))
          z0i = -c1(iFilt)*z1(i,iFilt) - c2(iFilt)*z2(i,iFilt)
     1         + g2(iFilt)*(y0i+2.*y1(i,iFilt)+y2(i,iFilt))
```

```
        u1(i,iFilt) = u0i
        y2(i,iFilt) = y1(i,iFilt)
        y1(i,iFilt) = y0i
        z2(i,iFilt) = z1(i,iFilt)
        z1(i,iFilt) = z0i
        data(iChan) = z0i
4000  continue
      endif
9999  return
      end
```

B.2.6 Subroutine *NotchF*—

```
      subroutine notchF (iFilt,aTime,data,reset)

c Richard Maine.  12 Aug 86.
c notch filter.
c u0,z0 are current in,out; u1,z1 previous time; u2,z2 two previous.
c Sample version based on x-29.

      implicit none

c********************* interface.
c iFilt(input): filter number.
c aTime(input): actual frame time.
c data (i/o): data vector for both input and output.
c reset(input): should filter be (re)initialized.

      integer iFilt
      logical reset
      double precision aTime,data(*)

c********************* common.
      integer maxInF,maxIch
      parameter (maxInF=10,maxIch=1000)
      common /filtCom/ iFilts,nFChs,nFChsU,fCh,fiCh,fChU,fiChU
      integer maxFF,maxFCh
      parameter (maxFF=2,maxFCh=50)
      integer iFilts(maxInF),nFChs(maxFF),nFChsU(maxFF),
     1  fCh(maxFCh,maxFF),fiCh(maxFCh,maxFF),fChU(maxFCh,maxFF),
     2  fiChU(maxFCh,maxFF)
      save /filtCom/

c********************* externals.
      intrinsic exp,cos

c********************* local.
c----- set appropriate break frequency.
      double precision freqRad
      parameter (freqRad=68.)

      integer iChan,i
      double precision wDt,w1Dt,dt(maxFF),
     1  b1(maxFF),c1(maxFF),c2(maxFF),g(maxFF),u0i,z0i,
     2  u1(maxFCh,maxFF),u2(maxFCh,maxFF),
     3  z1(maxFCh,maxFF),z2(maxFCh,maxFF)
      save dt,b1,c1,c2,g,u1,u2,z1,z2

      data dt/.005,.01/
```

```
c--------------------------- executable code ---------------------------

c********************* initialize filter at maneuver start.
      if (reset) then
c********** compute filter coefficients.
        wDt = freqRad*dt(iFilt)
        b1(iFilt) = -2.*cos(wDt)
        w1Dt = wDt*.707106781
        c1(iFilt) = -2.*exp(-w1Dt)*cos(w1Dt)
        c2(iFilt) = exp(-2.*w1Dt)
        g(iFilt) = (1.+c1(iFilt)+c2(iFilt))/(2.+b1(iFilt))
c********** initialize filter states.
        do 2000 i = 1 , nFChsU(iFilt)
          u0i = data(fChU(i,iFilt))
          u2(i,iFilt) = u0i
          u1(i,iFilt) = u0i
          z2(i,iFilt) = u0i
          z1(i,iFilt) = u0i
 2000   continue

c********************* filter.
      else
        do 4000 i = 1 , nFChsU(iFilt)
          iChan = fChU(i,iFilt)
          u0i = data(iChan)
          z0i = -c1(iFilt)*z1(i,iFilt) - c2(iFilt)*z2(i,iFilt)
     1         + g(iFilt)*(u0i + b1(iFilt)*u1(i,iFilt) + u2(i,iFilt))
          u2(i,iFilt) = u1(i,iFilt)
          u1(i,iFilt) = u0i
          z2(i,iFilt) = z1(i,iFilt)
          z1(i,iFilt) = z0i
          data(iChan) = z0i
 4000   continue
      endif
 9999 return
      end
```

References

1. *EMBOS User's Guide, Volume 1*, ELXSI, San Jose, California, 1983.

2. *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*, American National Standards Institute, New York, 1978.

Index to User's and Programmer's Guides

abbreviations 4

antonym 4

argument
- keyword 4
- positional 4
- switch 4

blanks 4, 15, 16

calculated signals 1, 12, 14–17, 21, 22

case 4, 15, 27, 28

combining data 9

command
- *copy* 6, 7, 9, 10, 20
- *do* 18
- *help* 4, 29
- *list* 17
- *method* 9, 13, 16, 22
- *quit* 7, 18
- *read* 6, 9, 12–14
- *show* 17
- *signals* 6, 9, 10, 14, 21
- *skew* 12
- *write* 6, 10, 11, 19

command file 18

comments 3, 18

continuation line 3, 9, 18

delimiters 3

dropouts 8

dt See parameter, *dt*

filter 12, 22

format 1, 19, 23, 25, 30

frame time 7, 9, 11–13, 16

interactive mode 2, 18

interpolation 1, 8, 9, 13, 16, 22, 23

limitations 2, 3, 8, 9, 12, 15, 16, 21, 23

merging 1, 9

multiple files 9, 10

parameter
- *dt* 7
- *fSkew* 12
- *nTimes* 8
- *thin* 7–9, 13

records 7

sample interval 7

sample rate 1, 7, 22

signal name 15
- duplicate 9
- renaming 9, 16

signal order 10

skew 1, 8, 11, 13, 16, 22, 23

splicing 9, 12

synonyms 4

syntax 3
- argument 4
- command 3
- parameter 4

system dependence 2–5, 7, 18–23, 25, 27

thinning See parameter, *thin*

time accuracy 8, 11, 13

time segment 1, 6

time skew 11

time tags 7

time tolerance 8

tolerance 8

UNIX 32

VAX 31

VMS 31

1. Report No. NASA TM-88288	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Manual for GetData Version 3.1—A FORTRAN Utility Program for Time History Data		5. Report Date October 1987
		6. Performing Organization Code
7. Author(s) Richard E. Maine		8. Performing Organization Report No. H-1403
		10. Work Unit No. RTOP 505-61
9. Performing Organization Name and Address NASA Ames Research Center Dryden Flight Research Facility P.O. Box 273, Edwards, CA 93523-5000		11. Contract or Grant No.
		13. Type of Report and Period Covered Technical Memorandum
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		14. Sponsoring Agency Code

15. Supplementary Notes

Contact author for information on program availability.

16. Abstract

This report documents version 3.1 of the *GetData* computer program. *GetData* is a utility program for manipulating files of time history data, that is, data giving the values of parameters as functions of time. The most fundamental capability of *GetData* is extracting selected signals and time segments from an input file and writing the selected data to an output file. Other capabilities include converting file formats, merging data from several input files, time skewing, interpolating to common output times, and generating calculated output signals as functions of the input signals.

This report also documents the interface standards for the subroutines used by *GetData* to read and write the time history files. All interface to the data files is through these subroutines, keeping the main body of *GetData* independent of the precise details of the file formats. Different file formats can be supported by changes restricted to these subroutines. Other computer programs conforming to the interface standards can call the same subroutines to read and write files in compatible formats.

17. Key Words (Suggested by Author(s)) Computer program Data processing Time history data	18. Distribution Statement Unclassified — Unlimited Subject category 61

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 145	22. Price* A07

**For sale by the National Technical Information Service, Springfield, Virginia 22161.*